

AD-A138 061

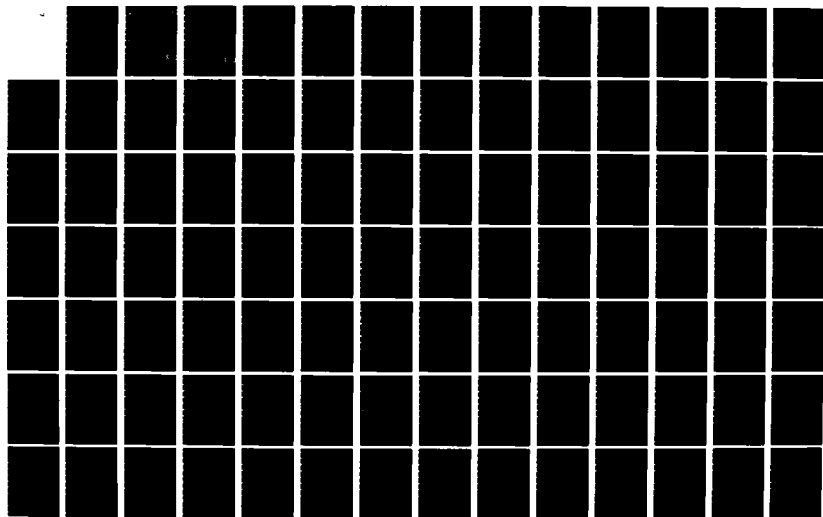
A GENERATOR OF RECURSIVE DESCENT PARSERS FOR LL(K)  
LANGUGES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING G B PAPROTY DEC 83  
AFIT/GCS/MA/83D-6

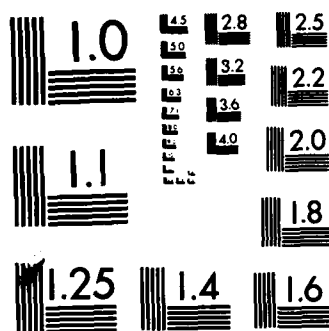
1/4

UNCLASSIFIED

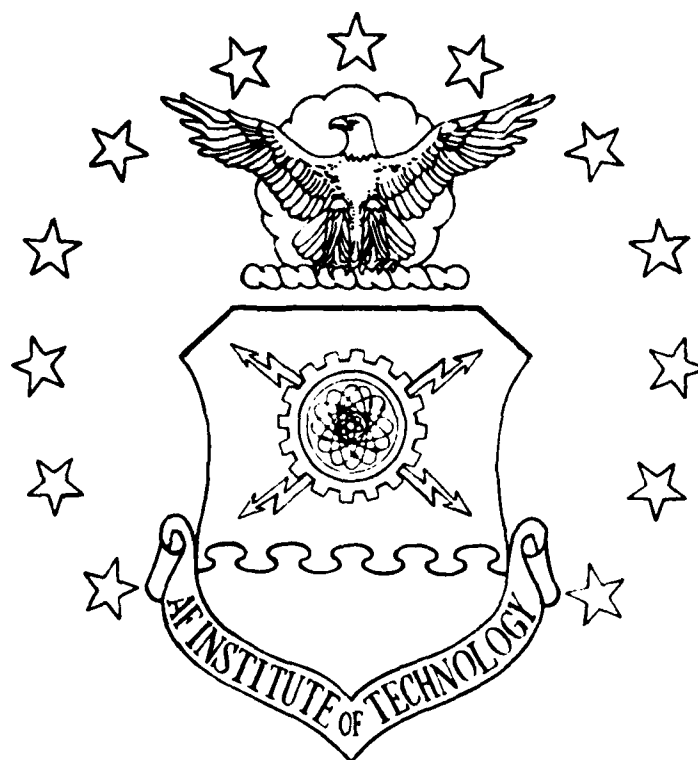
F/G 9/2

NL





AD A138061



LL -

A GENERATOR OF RECURSIVE DESCENT  
PARSERS FOR LL(K) LANGUAGES

THESIS

George B. Paprotny  
Capt USAF  
AFIT/GCS/MA/83D-6

DTIC  
ELECTE  
FEB 21 1984

S

D

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY

**AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

DIST

App

Lib

84 03 17 075

DTIC FILE COPY

AFIT/GCS/MA/83D-6

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
AI	



LL -

A GENERATOR OF RECURSIVE DESCENT  
PARSERS FOR LL(K) LANGUAGES

THESIS

George B. Paprotny  
Capt USAF

AFIT/GCS/MA/83D-6

DTIC  
ELECTE  
S FEB 21 1984 D

Approved for public release; distribution unlimited



AFIT/GCS/MA/83D-6

LL -

A GENERATOR OF RECURSIVE DESCENT  
PARSERS FOR LL(K) LANGUAGES

THESIS

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology  
Air University  
in Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science

by

George B. Paprotny, B.S.

Capt USAF

Graduate Computer Science

December 1983

Approved for public release; distribution unlimited

## Preface

This thesis has been the most monumental task that I have ever undertaken. However, because of that, and because I completed it and met my objectives, I have learned much about patience and perseverance. I have to thank my wife, Jan, for putting up with me through the months of torture that this thesis has brought. Without her smiling support, I never would have finished.

George B. Paprotny

## Table of Contents

	Page
Preface . . . . .	ii
List of Figures . . . . .	v
Notation . . . . .	vi
Glossary . . . . .	vii
Abstract . . . . .	ix
I. Introduction . . . . .	I-1
Thesis Justification . . . . .	I-2
Problem Statement . . . . .	I-3
Sequence of Presentation . . . . .	I-4
II. Compiler Theory Basics . . . . .	II-1
Compiling and Parsing . . . . .	II-1
Language Theory . . . . .	II-2
Parsers . . . . .	II-8
LL(k) and LR(k) Grammars . . . . .	II-8
Finite State Automata . . . . .	II-10
III. Standard Data Structures and Techniques . . . . .	III-1
Introduction . . . . .	III-1
Regular Expressions and Extended BNF . . . . .	III-1
Production Trees . . . . .	III-3
First and Follow Sets . . . . .	III-4
Parser Creation Algorithms . . . . .	III-7
IV. Requirements and Design . . . . .	IV-1
Introduction . . . . .	IV-1
Design Methodologies . . . . .	IV-1
Commenting . . . . .	IV-2
Requirements and Design Decisions . . . . .	IV-3
V. Input Format . . . . .	V-1
Introduction . . . . .	V-1
Relationship to YACC and LEX . . . . .	V-1
General Input Format . . . . .	V-2
Lexical Analyzer Definition . . . . .	V-3
Production Rules Format . . . . .	V-5
VI. Data Structures . . . . .	VI-1
Introduction . . . . .	VI-1
Finite State Automata (FSA) . . . . .	VI-2
Production Trees . . . . .	VI-2

	Page
Token Definition Structure . . . . .	VI-4
VII. Algorithms . . . . .	VII-1
Introduction . . . . .	VII-1
The Input Method . . . . .	VII-1
NDFSA to DFSA Conversion . . . . .	VII-4
Decorating the Production Trees . . . . .	VII-7
VIII. Parser and Lexical Analyzer . . . . .	VIII-1
Introduction . . . . .	VIII-1
General Structure . . . . .	VIII-1
The Lexical Analyzer . . . . .	VIII-2
The Parser . . . . .	VIII-4
The Values of Productions . . . . .	VIII-6
IX. Test Plan . . . . .	IX-1
Introduction . . . . .	IX-1
General Test Plan . . . . .	IX-2
Special Cases . . . . .	IX-3
The ADA Test . . . . .	IX-4
X. Conclusions and Recommendations . . . . .	X-1
Conclusion . . . . .	X-1
Recommendations . . . . .	X-2
Bibliography . . . . .	Bib-1
Appendix A: FIRST <sub>1</sub> and FOLLOW <sub>1</sub> Set Calculation . .	A-1
Appendix B: LL(1) Parser from Production Trees . .	B-1
Appendix C: Data Flow Diagrams . . . . .	C-1
Appendix D: Structure Charts . . . . .	D-1
Appendix E: Regular Expression Forms Used in LL . .	E-1
Appendix F: Production Rule Format . . . . .	F-1
Appendix G: FSA for Each Regular Expression Form .	G-1
Appendix H: Maximum Possible K Algorithm . . . . .	H-1
Appendix I: LL Output for a Partial ADA Grammar . .	I-1
Appendix J: LL User's Manual . . . . .	J-1
Appendix K: Code for LL . . . . .	K-1
Vita . . . . .	Vita-1

## List of Figures

Figure	Page
1. Example Production Trees . . . . .	III-4
2. Finite State Automata Data Structures . . .	VI-2
3. Production Tree Data Structures . . . . .	VI-3
4. Example FSA Creation . . . . .	VII-3
5. Non-Deterministic FSA Example . . . . .	VII-5
6. Empty Transition Removal . . . . .	VII-6
7. Conversion of Multiple Transitions . . . . .	VII-6
8. Last Multiple Transition Conversion . . . .	VII-7
9. After Deleting Inaccessible Transitions . .	VII-7
10. Decorate Production Tree Algorithm . . . . .	VII-8
11. Structure of the Lexical Analyzer . . . . .	VIII-3
12. Example of First Set Calculation . . . . .	IX-3
13. Possible Core Use Reductions . . . . .	X-4

## Notation

$\emptyset$	- the empty set
$\Sigma$	- the alphabet for a language
$\epsilon$	- the empty string, or 'is a member of'
$\cup$	- union
$\cap$	- intersection
$\rightarrow$	- x expands to y, in production rules, as in $x \rightarrow y$
$\Rightarrow$	- x expands to y, in the description of a parse
$\Rightarrow^+$	- x expands to y in 1 or more production rule applications
$\Rightarrow^*$	- x expands to y in 0 or more production rule applications
$ x $	- the number of tokens in the string x
$x^+$	- the set of all strings drawn from x of length $\geq 1$
$x^*$	- the set of all strings drawn from x of length $\geq 0$
$*$	- closure
$+$	- closure plus
$?$	- option
$\cdot$	- concatenation
$ $	- alternation

## Glossary

- BNF - (Backus Naur Form) a standard method of describing the productions in a grammar
- Code Generation - the process of creating executable code based on what tokens the parser has seen
- Compiler - a program that accepts a string of characters and processes them to determine whether the string exists in the language, and to perform some action (normally produce object code) based upon the string
- Deterministic - a process that always knows what to do based on the input
- EBNF - (Extended BNF) BNF with additional rules for describing a grammar more succinctly
- Grammar - a set of rules that defines the valid forms of a language
- LL(k) - used to describe a grammar which can be deterministically parsed top-down with a lookahead of k symbols
- Left Recursion - denotes a production which, when it is expanded, can end up with the production's non-terminal on the left hand side of the sentence created
- Lexical Analysis - the process of breaking input down into tokens for use by the parser
- Non-deterministic - a process that must backtrack when it discovers that its actions were incorrect based on the input
- Non-terminal - symbols in a grammar which define acceptable sentential forms, and are expanded to find acceptable strings of tokens
- Parsing - the process of accepting tokens and determining whether a string of tokens is acceptable in the language
- Production - one rule in a grammar
- Sentence - an acceptable string in a language
- Sentential Form - a string of terminals and non-terminals that occurs during a parse

Start Symbol - the non-terminal a compiler must begin with to perform a correct parse

String - a sequence of tokens


Strong LL(k) - describes a grammar for which a parser does not require knowledge of the complete sentential form to parse deterministically

Terminal - same as a token, except used in reference to a grammar

Token - one item in a language's alphabet




Abstract



A computer program was designed to generate a recursive-descent compiler for LL(k) languages. Grammar specification language was designed using Extended EBNF, along with a lexical analyzer specification language using regular expressions.

The program uses a well-tested but not formally proven method for determining the maximum possible k for a grammar. It then uses an iterative first and follow set calculation algorithm to determine the actual k for the grammar.



LL -  
A GENERATOR OF RECURSIVE DESCENT  
PARSERS FOR LL(K) LANGUAGES

I. Introduction

In some form or another, compilers have been in existence almost since the very first computer was created. They have been called assemblers, translators, and interpreters, but they all have had one purpose, which is to take a high level representation of a computer's instruction set and convert it through one or more steps into an executable program. The purpose of these compilers, along with the languages they represent, was, and still is, to decrease the programmers' workload by allowing a program to be created in a language more like English.

These representations of machine language have slowly evolved into a complex set of languages such as FORTRAN, COBOL, PASCAL, etc. Each one is different, and each needs its own compiler. Traditionally, each language has also needed a separate compiler for each machine, because each machine's instruction set is different. Thus, a whole new specialty within computer science has arisen, namely, compiler writing.

As will be seen in this thesis, most compilers can be set up in similar ways, and can have almost interchangeable

parts. Thus, as computers and language theory has progressed, we have been able to define standard techniques for representing language structures and for creating the compilers from those structures. In fact, the state of the art has progressed to the point that some portions of compilers can be automatically created for many languages.

Unfortunately, we have been unable to automatically create compilers for languages such as ADA. [DARPA, 1981] ADA is a highly complex language that has been in development since 1974, and even though it has been completely specified since 1981, only three compilers have been written for it so far. If a compiler generator could be created that would accept ADA, the compiler writing task would be simplified, and the cost of creating the compiler considerably reduced.

#### Thesis Justification

There is a relatively small number of compiler generators available on the market today; and, every one the author found in his extensive literature search [Gert, 1981] [Johnson, 1978] [Koster, 1971] [Leverett, 1980] has two major drawbacks, which are:

- a. Each one will create a compiler for those languages which only require the compiler to examine the next input in the source program. (ADA requires the compiler to look ahead two inputs.)

- b. The generators universally create compilers which consist of a small program with a very large array. This

array defines the specifics of the language. So, these compilers are almost unintelligible. As will be seen in the Compiler Theory Basics chapter, it turns out that the class of languages that the currently available compiler generators can use is called LR, and that a readable compiler can be created from the class of languages called LL. (An LL language which requires the compiler to look ahead  $k$  inputs is normally called an  $LL(k)$  language.)

There seems to be a gap in the software available to compiler writers. If a compiler generator could be created to examine the specification for a language and make portions of a compiler for it, regardless of the required lookahead, and if that compiler could be created in a manner that could be easily understood, (and modified if necessary), compiler writers would be far ahead of where they are now, and could concentrate on the problems that languages such as ADA create.

### Problem Statement

This thesis is concerned with developing software that will generate recursive-descent parsers for  $LL(k)$  grammars. This generator (called LL throughout this thesis) will accept a specification for a grammar, determine whether it is  $LL(k)$  for some  $k$ , and create the parser. It will also accept a specification for the lexical analyzer for that grammar, and create an analyzer that will operate as an integral part of the parser. Finally, the parser generator will accept user-supplied code fragments as part of the grammar, to allow the

user to process application-unique code during the parse.  
(Especially semantic analysis code.)

### Sequence of Presentation

The presentation of the material in this thesis (chapter by chapter) is as follows:

- a. Compiler theory basics.
- b. Standard data structures and techniques, with algorithms used in parts of the project.
- c. Design and requirements decisions, with justification for choice of design tools, and major project decisions.
- d. Input format, with justification for placement and format of each part of the input.
- e. Algorithms, with descriptions of the major algorithms used in LL.
- f. Data structures, with complete descriptions of all global structures used.
- g. Parser and lexical analyzer output, with descriptions of the data structures, support subroutines, and code used for each part of the output of LL.
- h. Test plan, with philosophy of testing, and general structure of testing procedures, along with descriptions of all special cases.
- i. The conclusion and recommendations, with a discussion of the project, and opinions of the author.

## II. Compiler Theory Basics

This chapter will define the terminology and theory needed in the chapters that follow. It will discuss compiling and parsing, language theory, parsers, grammars and finite state automata.

### Compiling and Parsing

Traditionally, a compiler is a program that translates a source program (written in some high-level language) into object code (some representation of the computer's machine language). The compilation process is usually considered to have three phases. These are: [Davie, 1981: 11-14]

1) Lexical Analysis. This first phase accepts the actual source program, and divides the character stream into units called tokens. A token may be a single character, but it is usually some sequence of characters. For example, "BEGIN", "(" and "PROCEDURE" are all tokens in PASCAL. Lexical analyzers also remove comments, spaces, and special control commands from the source input.

2) Parsing. The tokens generated during lexical analysis are passed directly to the parser, which must determine whether the program is syntactically and semantically correct. The parser usually follows some well-structured set of rules to determine correctness, and to structure the program for the code generator.

3) Code Generation. This phase accepts the structural output of the parser, and creates the object program. Generally, optimization on the object code is performed at various points in this phase. Some compilers mix the parsing and code generation to avoid the intermediate representation of the original program.

Lexical analysis is usually not discussed in compiler theory textbooks, because it is the easiest of the three phases to understand. Many compiler generators have some easy way to specify the rules for token creation.

Parsing, on the other hand, has a large body of theory devoted to it. Almost all of language theory applies exclusively to the syntactic analysis of programs. The other part of parsing, semantic analysis, has almost no developed theory. Therefore, existing compiler generators usually take care of syntax examination automatically, and leave any semantic work for explicit inclusion by the user.

The last phase of compilation, code generation, is the only one of the three that is still almost entirely an ad hoc process. Some work has been done in the specification of program semantics, but there are still many difficulties. [Leverett, 1980: 38-40]

### Language Theory

Before beginning any discussion of automatic compiler generators, a basic understanding of language theory is necessary.

Language Elements. The basic element of a language is the token. Tokens are analogous to letters in a natural language (i.e. English) since they are the basic building blocks of the language. Each language has a finite set of tokens, called its alphabet. The alphabet is usually designated  $\Sigma$ . Every source program is a sequence of tokens drawn from its language's alphabet. In this thesis, tokens will be represented by letters drawn from the beginning of the English alphabet (a,b,c etc.). [Gries, 1971: 15]

A string is a sequence of tokens drawn from  $\Sigma$ . The set of all possible strings of length 1 or more is  $\Sigma^+$ . The string of length zero (the empty string) is written  $\epsilon$ . The closure of  $\Sigma$ , or the set of all strings of length 0 or more, is  $\Sigma^*$ , which is equivalent to  $(\Sigma^+ \cup \epsilon)$ . Strings are usually represented by lower case letters drawn from the end of the alphabet (w,x,y, etc.). [Gries, 1971: 15]

Grammars and Languages. A language is some subset of  $\Sigma^*$ . This subset may be finite or infinite. Most programming languages have no bound on the length of strings, therefore they are infinite. A language must also have a structure. There must be a set of rules that determine the various ways that tokens from  $\Sigma$  may be organized to form strings in the language. This set of rules is called the language's grammar.

A grammar consists of terminals and non-terminals. A terminal is a member of  $\Sigma$ , and is the same as a token. A non-terminal serves as a placeholder in the grammar, and is



not in  $\Sigma$ . Each non-terminal stands for some set of strings in  $\Sigma^*$ . For example, consider the structure of telephone numbers in the United States. [Barrett, 1979: 18] A telephone number might be:

(212) 438-7021 (1)

This can be represented by the following structure

<area code> <exchange> <party> (2)

Each of these three also has its own structure. For example, the area code might be:

(<digit><digit><digit>) (3)

where a digit is one value from  $\{0,1,2,\dots,9\}$ . In this example, <area code>, <office>, <party>, and <digit> are non-terminals, and 0..9, dash, and parentheses are terminals. Throughout this thesis, non-terminals will be represented by a capital letter.

Grammars are represented by production rules, or productions, of the form  $x \rightarrow y$  where  $x$  and  $y$  are strings in a language's terminal and non-terminal set. [Aho, 1972: 85] For general productions,  $y$  may be any string of terminals and non-terminals, including  $\epsilon$ , and  $x$  may be any string of non-terminals and terminals, where there is at least one non-terminal in  $x$ .

These production rules are used to derive strings from other strings, that is, from

$w x z$  (4)

and applying  $x \rightarrow y$ , we derive

$w y z$  (5)

In order to specify the language completely, some starting string must be specified. If we could begin with any string at all, we would end up with  $\Sigma^*$  as the language. Thus, some string  $w$  is usually specified as the starting string. In fact, a single non-terminal is normally designated the start symbol, with no loss of generality. This is so because we can always have a production  $S \rightarrow w$  to begin a grammar.

When strings are derived in the language, the derivation process ends when no more non-terminals exist in the string. Thus, the terminal and non-terminal sets must be disjoint. Also, the reason for forcing all productions  $x \rightarrow y$  to have at least one non-terminal in  $x$  now becomes clear. We are not supposed to derive a new string from any string that consists of only terminals.

Now we have all the necessary ingredients for a formal definition of a grammar. A grammar  $G$  is a tuple  $(N, \Sigma, P, S)$ , where  $N$  is the set of non-terminals,  $\Sigma$  is the set of terminals,  $P$  is the set of productions, and  $S$  is the start symbol. [Barrett, 1979: 20]

Classes of Grammars. There are basically two grammar classes that take on the form described earlier. These are: context-sensitive, and context-free. [Barrett, 1979: 21-25]

The most general grammars are the context-sensitive grammars. These are characterized by productions of the form

$$x \rightarrow y \quad (6)$$

where  $x$  and  $y$  are elements of  $(\Sigma \cup N)^*$ , where  $x$  contains at least one member from  $N$ , and the length of  $x$  is less than or

equal to the length of  $y$  ( $|x| \leq |y|$ ). Note that this excludes any production of the form  $x \rightarrow \epsilon$ . An example of a context-sensitive grammar is: [Barrett, 1979: 20-21]

$$S \rightarrow aSBC \quad (7)$$

$$S \rightarrow abC \quad (8)$$

$$CB \rightarrow BC \quad (9)$$

$$bB \rightarrow bb \quad (10)$$

$$cC \rightarrow cc \quad (11)$$

and a possible derivation, with the rule used at each step, is

$$S \quad (12)$$

$$aSBC \quad S \rightarrow aSBC \quad (13)$$

$$aabCBC \quad S \rightarrow abC \quad (14)$$

$$aabBCC \quad CB \rightarrow BC \quad (15)$$

$$aabbCC \quad bB \rightarrow bb \quad (16)$$

$$aabbcc \quad bC \rightarrow bc \quad (17)$$

$$aabbcc \quad cC \rightarrow cc \quad (18)$$

The other class is the set of context-free grammars. These are the most common grammars, and the syntax for most programming languages is specified in some form of a context-free grammar. These grammars have productions of the form

$$A \rightarrow x \quad (19)$$

where  $A$  is a member of  $N$ , and  $x$  is an element of  $(\Sigma \cup N)^*$ . Note that in this case  $x$  may be  $\epsilon$ . A standard example of a context-free grammar is the following arithmetic expression grammar:

$$G = ( \{E,T,F\}, \{+,*,(,),a\}, P, E ) \quad (20)$$

where P is the set of productions

$$E \rightarrow E + T \quad (21)$$

$$E \rightarrow T \quad (22)$$

$$T \rightarrow T * F \quad (23)$$

$$T \rightarrow F \quad (24)$$

$$F \rightarrow ( E ) \quad (25)$$

$$F \rightarrow a \quad (26)$$

An example string, with a derivation starting at E, might be

$$E \quad (T+T) * F \quad E \rightarrow T \quad (27)$$

$$T \quad E \rightarrow T \quad (F+T) * F \quad T \rightarrow F \quad (28)$$

$$T * F \quad T \rightarrow T * F \quad (a+T) * F \quad F \rightarrow a \quad (29)$$

$$F * F \quad T \rightarrow F \quad (a+F) * F \quad T \rightarrow F \quad (30)$$

$$(E) * F \quad F \rightarrow ( E ) \quad (a+a) * F \quad F \rightarrow a \quad (31)$$

$$(E+T) * F \quad E \rightarrow E + T \quad (a+a) * a \quad F \rightarrow a \quad (32)$$

Sentential Forms. Derivations in a language are described in a manner similar to the material we have already discussed. One derivation step in a transformation is denoted

$$wxz \Rightarrow wyz \quad (33)$$

(Remember our original example, where  $x \rightarrow y$  was the production used.) In this example,  $wxz$  and  $wyz$  are sentential forms, if the derivation began with the start symbol.

[Gries, 1971: 20] One or more derivation steps is  $\Rightarrow^+$ , and zero or more is  $\Rightarrow^*$ . A sentential form that consists of only terminals is called a sentence.

From these definitions, we can define a language L which

is described by a grammar  $G$ , as [Barrett, 1979: 27]

$$L(G) = \{ x \mid x \in \Sigma^* \text{ and } S \Rightarrow^+ x \} \quad (34)$$

### Parsers

As previously mentioned, parsers determine the syntactic correctness of programs. There are two basic types of parsers: top-down and bottom-up. [Pyster, 1980: 34-71]

Top-Down Parsing. Top-down parsers attempt to recognize a sentence by beginning with the start symbol, and using the production rules to replace the non-terminals in the various derivation steps with the right-hand sides of the productions. A top-down parser uses a leftmost derivation. This means the non-terminals in the sentential form are replaced by their productions in a left to right order. Note that the example arithmetic expression derivation was leftmost.

Bottom-Up Parsing. On the other hand, bottom-up parsers accept symbols from the source program, and attempt to work from the sentence up to the start symbol. Thus, as symbols are collected, the right hand side of the sentential form is examined and replaced with a new non-terminal. This corresponds to a rightmost derivation (always replacing the rightmost non-terminal), in reverse order (bottom-up). Note that the parse is still being performed left to right.

### LL(k) and LR(k) Grammars

LL(k) and LR(k) grammars correspond to top-down and bottom-up parsing, respectively. [Berry, 1982: 17-19] Both

type of grammars are the largest subset of the context-free grammars for which a deterministic parser can be constructed for their parser type (with lookahead of k symbols).

[Aho, 1972: 333,371] A deterministic parser with lookahead is one that always knows which production to use at any stage in the parse, based upon the sentential form already created, and the next k symbols in the input.

It is interesting to note that, traditionally, LR (bottom-up) parsers have been written as table-driven programs. That is, the parser program is quite small, and serves only to manipulate the input and examine tables, which are generally represented as large arrays of integers. (See Aho [Aho, 1972: 368-396] for details on table-driven parsers.) LL parsers, on the other hand, are traditionally written as large groups of recursive procedures, where each procedure corresponds to one production in the grammar. Note that these recursive, top-down parsers are uniformly designated recursive-descent parsers. [Davie, 1981]

Formal LL(k) Definitions and Properties. This thesis is concerned with parser construction for LL(k) grammars. Therefore, some definitions of what constitutes an LL(k) grammar are now in order.

Formally, a grammar G is LL(k) if, for every production  $A \rightarrow x$  in G, [Barrett, 1979: 148]

$$S \Rightarrow^* wAy \Rightarrow wxy \Rightarrow^* wz... \quad (35)$$

and

$$S \Rightarrow^* wAy \Rightarrow wx'y \Rightarrow^* wz... \quad (36)$$

where  $|z| = k$  and  $z \in \Sigma^*$ , then

$$x = x' \quad (37)$$

There is an important theorem that involves LL(k) grammars, which is: An LL(k) grammar has no left-recursive non-terminals. [Barrett, 1979: 149-150] An example of a left-recursive grammar which fails to be LL(k) follows:

Let G have the productions

$$A \rightarrow Ax \quad (38)$$

$$A \rightarrow y \quad (39)$$

Then we have two derivations

$$S \Rightarrow^* wAz \Rightarrow^+ wAx^{k-1} z \Rightarrow wAx^k z \Rightarrow wyx^k z \Rightarrow^* wr... \quad (40)$$

and

$$S \Rightarrow^* wAz \Rightarrow^+ wAx^{k-1} z \Rightarrow wyx^{k-1} z \Rightarrow^* ws... \quad (41)$$

where  $x^i$  means  $xxx...x$ , with  $|xxx...x| = i$ . It is easy to see that if  $|r| = |s| = k$ , then  $r = yx^{k-1}$  and  $s = yx^{k-1}$ , so  $r = s$ , but the two derivations are different. This results in an ambiguity.

### Finite State Automata

Definition. A finite state automaton (FSA) is a system which has states, and transitions between those states.

Based on an input, a FSA will move between its states along the transitions. A FSA is always considered discrete, that is, it is always in exactly one state, or is moving through exactly one transition to another state. [Barrett, 1979: 63]

A finite state automaton has four parts. These are:

- a. A start state
- b. A set of intermediate states
- c. A set of ending states
- d. Transitions between states

This simple apparatus can recognize input defined by regular expressions, [Barrett, 1979: 104-110] which are defined extensively in Chapter III. Since regular expressions can easily define the tokens for a language, this will be important to LL.

Non-Deterministic vs Deterministic FSA. A FSA is deterministic when no choices are provided in any of its moves. Each move is completely specified by the current state and the next input. A non-deterministic finite state automata (NDFSA) is one in which arbitrary choices are permitted in some of its moves. Therefore, a NDFSA is forced to backtrack if it can not continue its moves, because it may have made an arbitrary choice, and must go back to take a different choice.

A NDFSA is defined exactly the same as a deterministic FSA (DFSA) with two additions. A NDFSA may have multiple transitions on the same input from the same state, and it may have empty transitions, that is, a transition where no input is processed.

FSA Representation. In this thesis, FSA are diagrammed as follows:

- a. States are circles containing state numbers.



b. Directed arcs from one state to another are transitions. Each arc will have a character or  $\epsilon$  (empty) marking it.

c. The start state will be state 1. Also, no start state will be an ending state.

d. The ending states will have a double circle. If some end state has an action that must be performed, there will also be an action number in parentheses.

### III. Standard Data Structures and Techniques

#### Introduction

There have been many tools and techniques relating to compiler construction described in the literature. This chapter will describe some of these techniques.

First, regular expressions and extended BNF will be discussed. Then, production trees and their construction will be defined, followed closely by First and Follow Set construction using these trees. And last, an algorithm for automatic LL(1) parser construction will be described.

#### Regular Expressions and Extended BNF

Backus-Naur Form. Backus-Naur Form (BNF) is a practical method of specifying a grammar. [Davie, 1981: 39] In it, the symbol  $\rightarrow$  is replaced by  $::=$ , non-terminals are surrounded by "<" and ">", and two productions such as  $A \rightarrow x$  and  $A \rightarrow y$  can be represented as  $A ::= x \mid y$ . (The symbol  $\mid$  is used as "or".) For example, the arithmetic expression grammar specified in the Language Theory Section of Chapter II might be represented in BNF as

$$\langle E \rangle ::= \langle E \rangle + \langle T \rangle \mid \langle T \rangle \quad (42)$$

$$\langle T \rangle ::= \langle T \rangle * \langle F \rangle \mid \langle F \rangle \quad (43)$$

$$\langle F \rangle ::= ( \langle E \rangle ) \mid a \quad (44)$$

Regular Expressions. A regular expression is a compact way of representing the grammar of a regular language.

[Barrett, 1979: 100] Regular languages are a subset of the context-free languages, and are represented by productions of the form

$$A \rightarrow aB \text{ where } B \in N, \text{ and } a \in \Sigma \quad (45)$$

and

$$A \rightarrow b \text{ where } b \in \Sigma \quad (46)$$

Regular expressions are quite formal, and have three basic operations. These are concatenation, alternation, and closure. Parentheses may also be used. In the following definitions, E and F are regular expressions, x and y are strings, and X and Y are sets of strings.

a. Concatenation is represented by placing two regular expressions next to each other. EF is the concatenation of E and F, and is formally

$$EF = \{ x \mid x \in E \text{ and } y \in F \} \quad (47)$$

b. Alternation is represented by | or +. (This thesis will always use |.) E | F is the alternation of E and F, and is formally

$$E \mid F = \{ x \mid x \in E \text{ or } x \in F \} \quad (48)$$

c. Closure is represented by {E}, and is formally

$$\{ E \} = \{ xY \mid x \in E \text{ and } Y \in \{E\} \} \cup \{\epsilon\} \quad (49)$$

d. Parentheses may surround any part of a regular expression to change the way the expressions are handled. For example, ab|bcd is equivalent to (ab)|(bcd), not a(b|bc)d.

The symbols {, }, (, ), and | are all in the regular

expressions, and therefore can not be used as part of any language a regular expression represents. A solution to this might be to surround any occurrences of these characters in the language by quotes.

Extended BNF. Regular expressions and BNF have been combined in a relatively straightforward way to come up with a more concise grammar specification language called Extended BNF (EBNF). [Gert, 1981: 2-6] Unfortunately, there is no standard for EBNF, but the methods in use are similar enough to avoid confusion. A common one is described here.

The symbols [ and ] are used in EBNF to show that a particular string is optional; i.e. it can occur 0 or 1 times. Then, the EBNF production

$$\langle A \rangle ::= a[bc]d \quad (50)$$

is equivalent to the BNF production

$$\langle A \rangle ::= abcd \mid ad \quad (51)$$

The symbols { and } are used to show that a string can occur 0 or more times. Then, the EBNF production

$$\langle A \rangle ::= a\{bc\}d \quad (52)$$

is equivalent to

$$\langle A \rangle ::= a\langle B \rangle d \quad (53)$$

$$\langle B \rangle ::= bc\langle B \rangle \mid \epsilon \quad (54)$$

### Production Trees

Barrett [Barrett, 1979: 178-179] has a method of representing productions that can be useful in parser construction.

He represents EBNF as production trees.

Each production tree in a set represents one non-terminal A. The interior nodes are from the set  $\{ \cdot, |, *, ? \}$ , where  $\cdot$  is concatenation,  $|$  is alternation,  $*$  is closure, and  $?$  is option. Since  $\cdot$  and  $|$  are binary operations, each has two children in a tree. But  $*$  and  $?$  are unary operations, and therefore each has one child. The leaves of the tree are from the set  $N \cup \Sigma \cup \{\epsilon\}$ .

As an example, the grammar G with productions

$$S \rightarrow a\{B\}d \quad (55)$$

$$B \rightarrow [b]c \mid bb \quad (56)$$

can be represented by the production trees in Figure 1.

### First and Follow Sets

When the definition of LL(k) grammars was given earlier in this thesis, we saw it was: A grammar G is LL(k) if and only if for every production  $A \rightarrow x$  in G,

$$S \Rightarrow^* wAy \Rightarrow wxy \Rightarrow^* wz... \quad (57)$$

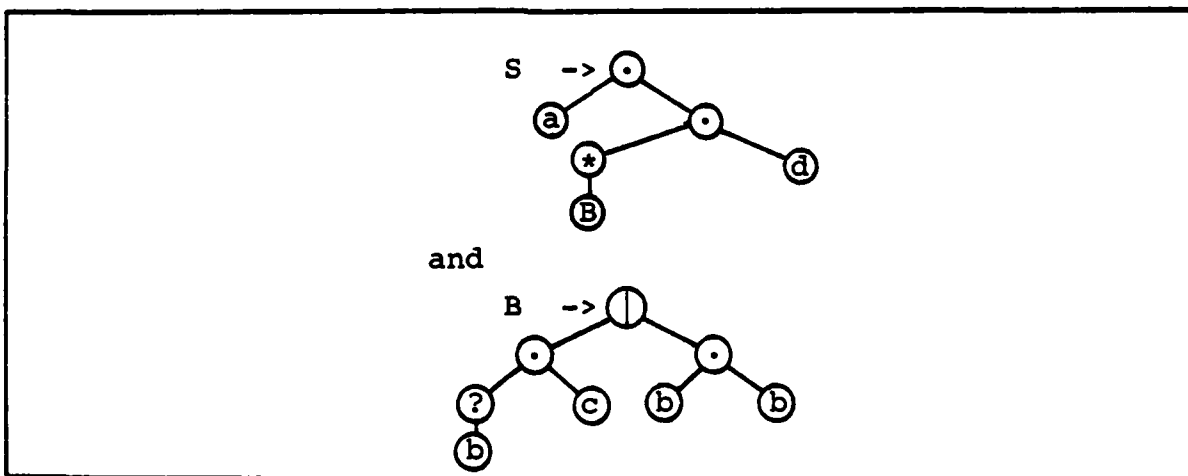


Figure 1. Example Production Trees

and

$$S \Rightarrow^* wAy \Rightarrow wx'y \Rightarrow^* wz... \quad (58)$$

where  $|z| = k$  and  $z \in \Sigma^*$ , then

$$x = x' \quad (59)$$

This definition does not appear to be easy to implement in an algorithm, so it might not be useful for reading a grammar and determining if the grammar is  $LL(k)$ . But, first and follow sets appear to solve this problem.

The  $FIRST_k$  set of a string  $w$  is defined as [Aho,1972:300]

$$\{ x \mid w \Rightarrow^* xy \text{ and } |x| = k \text{ or } w \Rightarrow^* x \text{ and } |x| < k \} \quad (60)$$

and the  $FOLLOW_k$  set of a string is defined as [Aho, 1972: 343]

$$\{ x \mid S \Rightarrow^* vwy \text{ and } x \in FIRST_k(y) \} \quad (61)$$

With these definitions, we can define an  $LL(k)$  grammar as follows: [Barrett, 1979: 153]

A grammar  $G$  is  $LL(k)$  if and only if, for every pair of productions

$$A \rightarrow u \text{ and } A \rightarrow v \quad (62)$$

and every leftmost sentential form  $wAx$  derivable in  $G$ ,

$$FIRST_k(ux) \cap FIRST_k(vx) = \emptyset \quad (63)$$

Note that for this sentential form,  $x$  is  $FOLLOW_k(A)$ . This means that the definition can be restated as follows:

$G$  is  $LL(k)$  if and only if for every pair of productions

$$A \rightarrow u \text{ and } A \rightarrow v \quad (64)$$

and every leftmost sentential form  $wAx$  derivable in  $G$ ,

$$FIRST_k(u FOLLOW_k(A)) \cap FIRST_k(v FOLLOW_k(A)) = \emptyset \quad (65)$$

This definition of  $LL(k)$  is only valid for each sentential form  $wAx$ . It is not true for all forms of that type. An example of this is the following grammar  $G$  with productions

$$S \rightarrow aBbc \mid Bc \quad (66)$$

$$B \rightarrow b \mid c \mid \epsilon \quad (67)$$

In this grammar,  $S$  derives  $aBbc$  and  $Bc$ . For  $aBbc$ , the possible sentences in  $G$  are  $abbc$ ,  $abcc$ , and  $abc$ . So, for  $aBbc$ ,

$$FIRST_2(b \text{ FOLLOW}_2(B)) = bb \quad (68)$$

$$FIRST_2(c \text{ FOLLOW}_2(B)) = cb \quad (69)$$

$$FIRST_2(\epsilon \text{ FOLLOW}_2(B)) = bc \quad (70)$$

which are all pairwise disjoint. For  $Bc$ , the possible sentences in  $G$  are  $bc$ ,  $cc$ , and  $c$ . So, for  $Bc$ ,

$$FIRST_2(b \text{ FOLLOW}_2(B)) = bc \quad (71)$$

$$FIRST_2(c \text{ FOLLOW}_2(B)) = cc \quad (72)$$

$$FIRST_2(\epsilon \text{ FOLLOW}_2(B)) = c \quad (73)$$

which are also pairwise disjoint. However, if the calculations were performed without regard to the leftmost sentential form, the six  $FIRST_k$  sets would not be disjoint, because the string  $bc$  occurs in two different calculations. [Barrett, 1979: 155]

A grammar which has disjoint  $FIRST_k$  sets without considering the leftmost sentential form is defined to be strong  $LL(k)$ . The difference between a strong  $LL(k)$  grammar and an  $LL(k)$  grammar will become important when generating a parser for the grammar. Succinctly stated, it is easier to look

ahead  $k$  symbols than it is to look ahead and look behind to see what the leftmost sentential form is.

### Parser Creation Algorithms

There are two published algorithms that are useful for creating recursive-descent parsers from EBNF. The first is  $FIRST_1$  and  $FOLLOW_1$  set calculation on production trees, and the second is construction of recursive-descent LL(1) parsers from the production trees decorated with first and follow sets. (In the rest of this chapter, FIRST and FOLLOW is used instead of  $FIRST_1$  and  $FOLLOW_1$  if no ambiguity will result.) [Barrett, 1979: 180-187]

FIRST and FOLLOW Set Construction. The algorithm for FIRST and FOLLOW set calculation is a multi-pass process. In it, the FIRST sets are completely calculated, and then the FOLLOW sets are calculated. Each set of calculations will be performed until no changes are observed in the FIRST or FOLLOW sets for any node on the production trees. Since the entire algorithm is in Appendix A, only one example is given here.

To calculate the FIRST set of the node | (alternation), calculate the FIRST sets for the right and left children of the node, and then add the FIRST sets of both children to the FIRST sets of the alternation node. To calculate the FOLLOW set for the same node, add the FOLLOW set of the node to the FOLLOW set of both children, and then calculate the FOLLOW sets of the children.



Parser Construction for LL(1) Grammars. Parser creation is the final phase of the compiler construction process. The complete algorithm is described in Appendix B. This process assumes an unambiguous LL(1) grammar. The parser is created using the production trees that have been decorated with FIRST and FOLLOW sets. There is no error recovery in this created parser, and, when executed, it will stop as soon as an input error is discovered. As given, the procedure is rudimentary, but it is an excellent first step towards creating a more complex parser.

## IV. Requirements and Design

### Introduction

In any software development project, the development process is very important to the success of the project. This chapter will first describe the software engineering tools chosen by the author. Then, some philosophies on commenting will be discussed, and last, the requirements and design decisions will be detailed.

### Design Methodologies

There are innumerable techniques in use today for designing and developing computer software. Choosing one or more techniques for a project such as LL, which is designed, developed and implemented by one person, is a very introspective decision. The author believes that the technique chosen must be flexible enough to accept any type of project, be formal enough to force a good design process, while at the same time be easy enough to use so that time is not spent on excessive detail. Also, the technique chosen should create paperwork that will be very useful as maintenance documents, and be understandable enough so that another relatively experienced programmer can learn how the system works.

To satisfy these requirements, the author has chosen Data Flow Diagrams (DFDs), [Peters, 1981: 139-140,145] and Structure Charts (SCs). [Peters, 1981: 60-62] The DFDs were

used at the higher levels of the design phase. They assisted in breaking down the complex structure of the project into manageable, understandable pieces. At each stage of the DFDs, the author stopped breaking down the pieces when he had a sub-system that he could understand completely. At that point, SCs came into play.

The SCs were used to cut the pieces into multiple, stand-alone modules. The criteria for breaking a task into modules were:

- a. Does each sub-task stand by itself?
- b. Is the interface with the calling routine fairly small?
- c. Is the task large enough to warrant its own module?

The DFDs and SCs in Appendices C and D were developed with all of the above in mind. It should be noted that the format is not formal, because each diagram was written to make the meaning of the diagram as clear as possible, while at the same time maintaining a high level of accuracy.

#### Commenting

DFDs and SCs are extremely valuable in all phases of a large project. However, nothing can replace in-code comments and program headers to completely round out the information available to the reader.

Program headers in LL consist of all pertinent information about each module. This includes the module's purpose, interface, variable list, and basic algorithm, along with

other valuable information. Comments are placed in the code to enhance all the other information that already exists. The author used the philosophy that having too many comments is a very poor programming practice, since readers may become just as frustrated with meaningless comments as with no comments.

### Requirements and Design Decisions

The purpose of this thesis is to create an LL(k) parser generator along with a lexical analyzer. Before progressing into the design stage, some decisions needed to be made, and specific requirements determined. This section lists and justifies these decisions and requirements.

Use of AFIT VAX 11/780 (SSC). The lack of a sponsor to this thesis severely limited the choice of computers. Excluding microcomputers, which are not reasonable for a project of this type, there are only three; the AFIT VAX 11/780 (Scientific Support Computer), the AFIT VAX 11/780 (database research), and the ASD CDC 6600. The ASD CDC 6600 is no longer used by most AFIT students, (because of its poor response and "unfriendly" operating system), and is therefore unreasonable for this project. Also, the database research computer has very limited disk space, and is frequently unavailable for use. The SSC is the "standard" student computer, and has enough disk and memory to satisfy any LL's requirements, therefore it became the computer of choice.

Use of C Programming Language. During the design and

implementation stage, many data structures were built and manipulated, and therefore LL will require a pointer-oriented language to give maximum flexibility to the code. C and PASCAL were the only two languages available that have pointers. Each one is a high-level language, but C can also be used at the bit level to manipulate data, and is the language that most of the systems that exist on the SSC are written in. Even though C is more cryptic than PASCAL, its advantages far outweigh this single disadvantage.

Use of YACC and LEX. YACC (Yet Another Compiler Compiler) [Johnson, 1978] and LEX (LEXical analyzer creator) [Lesk, 1978] are both resident on the SSC. The use of a YACC/LEX input processor alleviates the need for a massive amount of coding for reading in the grammars, processing them, and recognizing what all the strings mean. It is a standard tool used by many other systems, and so is proven effective. Also, as we will see later, the YACC input can be massaged to create an input to LL as a test case. Therefore, YACC and LEX were chosen to process the input.

Use of C for Output Parser. C is used in the output for all the same reasons that was used for LL itself.

Interface Between LL Parser and Lexical Analyzer. The interface between the parser and lexical analyzer is important at this point because one of the first tasks in design is to create the input formats and data structures, and this interface will have an effect on them. Also, since LEX already

exists, it would be wise to allow the use of LEX rather than the LL lexical analyzer if the user chooses. Therefore, it is desirable that the interface between the parser and lexical analyzer be as close to the YACC/LEX interface as possible.

Other Requirements. The rest of the requirements are not directly justified, but are either obvious, or are personal preferences that have little effect on the objective. These are:

a. The output parser and lexical analyzer should be as simple and easy to understand as possible.

b. Both the lexical analyzer and parser should be non-backtracking. That is, they should be able to make decisions deterministically.

c. The lexical analyzer should be a deterministic finite state automata (DFSA). (This is the only method known to the author for performing this task.) Aho [Aho, 1972: 113-137] is a valuable reference to understand FSA.

d. The CPU, memory, and disk requirements of LL are not important, unless they become totally unmanageable. (This alleviates the need to make very compact and complex data structures, thereby increasing the project's simplicity.)

e. For practical reasons, there must be a facility to allow the productions to have a value, and to allow actions and other elements of a production to have values. The user should be able to retrieve and store these values, in a manner similar to that of YACC.

## V. Input Format

### Introduction

This chapter discusses the decisions made during the initial design phase, specifically the reasons for the input format. First, the general outlines of the format will be discussed. Then, the specific design decisions for the lexical analyzer will be given, and finally the format of the production rules will be discussed.

### Relationship to YACC and LEX

As indicated in the previous chapter, the decision to use YACC and LEX to accept the input to LL was made during the first days of design. However, this decision was not made strictly to decrease the programming workload.

Both YACC and LEX are very similar to the products that are contained within LL. YACC is a parser generator, and LEX is a lexical analyzer creator. It became easy to see that it should be possible to take the input definitions written for YACC, and modify them so they would input easily to LL as a test case. Thus, the format of the input to LL became important, because any similarity to the input format of YACC would be advantageous in that conversion. (This similarity would also provide YACC users an easy transition to LL use.)

The input format for the LL lexical analysis portion became similar to LEX for reasons that were much the same.

Because the purposes of the two lexical analyzers are the same, namely, to return tokens from the input stream, the decision was made to keep the two as generally similar as possible.

#### General Input Format

At this point, it was necessary to formalize the input formats, determine where to place variable and subroutine declarations, and decide whether to force the user to declare the names of tokens.

Placing the Variables and Subroutines. There were two considerations as to placing the variables and subroutine declarations. One, what effect would the placement have on the user, and two, what effect would it have on the ease of development. For the user, placing them at the beginning or the end seemed to be best. For the ease of writing the YACC and LEX input routines, any location was acceptable, but the end seemed to be the best, because no other types of input would be expected, and the processing of the declarations would be merely a copying operation. The deciding factor was, once again, the format of YACC. It places these types of inputs at the end, and so LL also places them at the end.

To Declare or Not to Declare Tokens. Whether or not to force the user to explicitly declare token names was a more difficult and complex decision. To a sophisticated user, placing all of the token names in a separate list is an unnecessary task, since every name in a grammar that is not



defined as a production is automatically a token. However, it is easier to detect such things as spelling errors and forgotten production rules when the token names are specified beforehand, and the less sophisticated user will appreciate that. After considering these arguments, the decision was made to force the user to declare tokens specifically. The time required to key in the token names is not so great as to outweigh the advantages (to the user) of having extra error checking capability.

Final Input Format. All the input parts have been discussed, and now all that is left is to show the order of the parts. Again, YACC has been used as a model. In it, the tokens are defined at the beginning. Since the token names are used throughout the rest of the grammar, this is a good practice, and was adopted for LL. The lexical analysis output has been placed second, and the production rules placed third. All these are followed by the variable and subroutine declarations. The delimiters between the parts are the same as YACC's, specifically "%%".

#### Lexical Analyzer Definition

As stated earlier, this part of the input will look very much like the input to LEX. The Requirements and Design Chapter defined the necessary parts of this input to be: (1) the definition of the character sequences that make up tokens, and (2) the definitions of the actions to be performed when a specific character sequence is recognized. (Because of the

interface requirements between the LL parser and lexical analyzer, each character sequence that defines a token must have, at a minimum, a "return (token-name)", since the lexical analyzer is a subroutine.)

The following defines the basic structure of the lexical input:

- a. Token definitions begin in column 1.
- b. Token definitions are terminated by a space, tab or newline.
- c. Actions consist of a sequence of C statements.
- d. Actions begin after the token definition, and can be continued on subsequent lines.
- e. A line beginning with a space, tab, or newline is part of the action for the immediately preceding token definition.
- f. A token definition need not have an action.

Structure of the Token Definitions. In LEX, tokens are defined using a modified form of regular expressions. (Regular expressions are defined in Chapter III.) These regular expressions are an easy way to represent token definitions, and actual regular expressions can be converted into a deterministic finite state machine.

The LL lexical analyzer will use the standard regular expression forms, as described in Appendix E. These do not include the following LEX forms:

- a. '^'. In LEX, this character is used to denote the

beginning of a line; i.e. the regular expression only succeeds if a beginning of line is found at the point of the carat.

b. '\$'. This character denotes the end of a line, in a manner similar to (a).

c. '<...>'. Denotes a start state. [Lesk, 1976: 48]

d. 'x{m,n}'. Denotes m through n occurrences of 'x'.

None of these are standard regular expression forms, so they are not included in the LL lexical analyzer input.

#### Production Rules Format

The format for the production rules is similar to EBNF, with one addition, which is "closure plus". Closure plus is the same as closure, except that the expression inside the braces will be repeated one or more times, rather than zero or more times.

The complete definition of the format can be found in Appendix F, but the following list gives the highlights of the changes and additions from EBNF, along with the reasons for each one:

a. The first production rule defines the starting symbol. If the starting non-terminal was generalized by having it be any of the non-terminals, a new input to LL would be needed to define that symbol. As it is, no generalization is lost, since any grammar can be modified to move the starting production S to the beginning, or to make a new production  $\text{NewS} \rightarrow S$ .

b. Multiple production rules may define the same non-terminal. This rule allows an extra measure of generality LL would not otherwise have had.

c. Non-terminal names may be surrounded by < and >, as long as all instances of the name are surrounded. This allows a user to separate terminals and non-terminals with a quick glance, and also allows grammars that come from technical manuals to be copied directly. (Often the non-terminals have these surrounding characters.)

d. Every production rule ends with a semicolon. This rule allows LL to perform more accurate error detection, and provide the user with more accurate feedback if an error is found.

e. Actions begin with '=' and end with '}'. These actions need to be set off by some delimiter, and { } was already taken, because of closure. They would have been best, since they also delimit C statement blocks. The character '=' was chosen as the prefix to '{' because of the proximity of the two characters on the keyboard of the computer system that LL was created on.

f. "Closure Plus" is defined by { expression }+. Note the + to signify the difference from closure. This is an extension from regular expressions, where "one or more" is also represented with a +.

## VI. Data Structures

### Introduction

After the initial design decisions have been made, and what is to be done has been determined, the next step is to determine how it is to be done. For a project such as LL, which must maintain a large amount of data, manipulate that data, and finally create output from the data, the data structures involved are vital.

For this project, two considerations were used in building the required data structures. First, they should be easy to build, to use, and to modify if necessary. These are desirable attributes in any project, but especially in one which is a prototype. And second, these data structures should correspond as closely as possible to the generalized data structures as described in Chapter III. This makes them easy to understand by someone studying LL, and a direct relationship can be drawn between the theory and the implementation.

This chapter describes three sets of data structures. First, those dealing with the lexical analyzer/finite state automata will be discussed. Second, the structures for the parser/production trees will be described, and third, the token definition structure will be detailed.

### Finite State Automata (FSA)

The data structure for FSA will exactly model the theoretical FSA as described in the Finite State Automata Section of Chapter II. Figure 2 shows the format of this data structure. The reader must remember that for those FSA which are non-deterministic, a particular character may have multiple transitions from one state to other states, so the data structure is not quite as easy to comprehend as it might otherwise be.

### Production Trees

The data structure for the production trees is more complex, but that is because there is more data involved. There are basically two sets of data. First, the production trees themselves must be represented. (Chapter III describes pro-

#### State Data Structure

1. Action-id ( >0 if end state. Points to a particular action. =0 if not end state)
2. State number
3. Pointer to next state (connects states together)
4. Pointer to first empty transition
5. Array of 128 pointers to first transitions, one for each of 127 possible characters, and also one for the character NUL, which = 0. (Since each letter has its own ASCII code, the code is used to enter the array.)

#### Transition Data Structure

1. Points to state which the transition is to
2. Pointer to the next transition, if there is more than one transition on this character from this transition's state

Figure 2. Finite State Automata Data Structures

duction trees in detail.) It is important to note that the structure must be flexible enough to allow for nodes with zero, one, or two children. Second, the first and follow sets for nodes must be represented. Since these may be of any length, the data structure must be flexible enough to handle it. Figure 3 shows the data structures that have been designed.

#### Production Tree Head

1. Pointer to next production tree head
2. Name of non-terminal
3. Pointer to top of tree

#### Tree Node

1. Node type (closure, alternation, etc.)
2. Pointer to first set
3. Pointer to follow set
4. Pointer to left child if binary node, only child, if unary node
5. Pointer to right child if binary node, or a unique reference number if the node is a non-terminal
6. Action number, if the node is an empty node because an action is required

#### First/Follow Node

1. Pointer to next first/follow node
2. Pointer to first token node for this first/follow item

#### Token Node

1. Pointer to next token node in this first/follow item
2. Pointer to a terminal definition node

Figure 3. Production Tree Data Structures

There are two non-trivial parts of the production tree data structure that should be discussed at this point. First, a non-terminal node needs a reference number because of the requirement to remember the leftmost sentential form, which is shown in the First and Follow Sets Section of Chapter III. With the reference number, which would be unique for each reference of a non-terminal, references to the follow set of a non-terminal can be made distinct based on which production rule referenced that non-terminal. And second, a method of placing actions into the production tree is necessary, so empty nodes are created whenever actions are found, and an action identification number is placed in the node. Of course, those empty nodes which are part of a grammar will not receive an action number.

#### Token Definition Structure

This final data structure is the simplest one in LL. It consists of a token name, a token number, and a pointer to the next terminal in the list. The token number will be used to identify each token, without resorting to the character string.



## VII. Algorithms

### Introduction

The two previous chapters in this thesis define the input format and the data structures for the LL project. Once both of these were defined, creating the algorithms for the project was a relatively straightforward task. Almost all of these algorithms were either in a book, or were very similar to other common algorithms. The only major difficulty occurred in decorating the production trees for a grammar.

The algorithms will be described in the following order:

- a. the input method, to include the creation of the NDFSA and the production trees
- b. converting the NDFSA to a DFSA, and outputting the appropriate C code for the lexical analyzer
- c. decorating the production trees

### The Input Method

As we saw in the introduction to Chapter V, the input consist of four parts. These are the token definitions, the regular expressions, the production rules, and the variable and subroutine declarations. This section shows how YACC was used to process these inputs and perform the required actions. At this time it should be noted that some understanding of YACC is necessary to fully understand this section. [Lesk, 1976] Also, note that each time output is

mentioned, the format and reasons for the output is not clear. These problems will be cleared up in the next chapter.

The Token Definitions. Two actions were required to input the token definitions. The first was to create a list of token names and their corresponding integer values for use during the production input, and the second was to output a C "define" statement for each token to be used in the created parser. This task required only one production rule, and was straightforward.

The Regular Expressions. Processing the regular expressions and creating the NDFSA required productions very much like those found in Appendix E. There were essentially two tasks involved here. The first one involved creating and linking states at the appropriate places in the production rules, and the second involved accepting the actions and outputting them, along with appropriate surrounding code.

An example of the method used to link the states is included at this point. This example will show how multiple alternatives would be linked together to form a sub-automata. Assume that the following YACC production rule exists:

sub\_FSA : alternative | sub\_FSA (74)

and that the input is

ab|cd|e (75)

Each of 'ab', 'cd', and 'e' are alternatives. Assume that as each alternative is parsed, the appropriate sub-FSA is returned. The alternative 'ab' returns as in Figure 4(a).

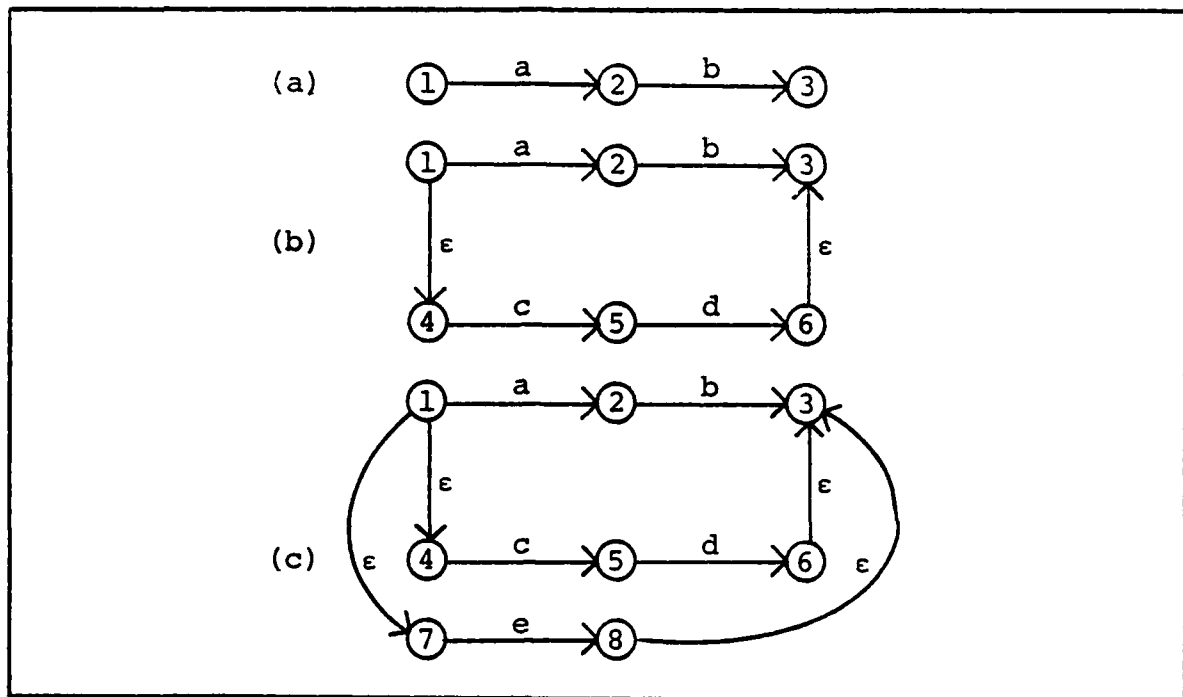


Figure 4. Example FSA Creation

So, the task is to accept the three alternatives and meld them into one sub-FSA. When the FSA for 'cd' is returned, two empty transitions are added, and the FSA is as shown in Figure 4(b). When 'e' has been parsed, two empty transitions are again added, and we have the FSA in Figure 4(c).

These empty transitions are added in the actions for the regular expression productions. The FSA in Figure 4(c) will accept 'ab', 'cd', or 'e' as input. That FSA is non-deterministic. (See Appendix G for a complete list of all the structures that are created for each type of regular expression.)

Ending states have an action number attached to them.

When the action is parsed by LL, it is immediately output with the surrounding code, and the ending state is marked.

The Production Rules. The production rules are parsed in much the same way as the regular expression, so no example will be given here. The general scheme, however, is to create the trees from the bottom up, and for each production rule to accept one or two subtrees, create a parent node, and then to pass the new tree up to the next production rule.

When actions are parsed, an empty node is created, and it is marked with an action number. The actual code is output, along with appropriate surrounding code, which uses the action number as part of the identification for the action.

The Variable and Subroutine Declarations. These declarations were the easiest to parse. All that was required was to accept the code and output it with no changes.

#### NDFSA to DFSA Conversion

One of the requirements of the project is to avoid all types of backtracking. Therefore, the FSA must be deterministic. The following algorithm to convert a NDFSA to a DFSA has been proven effective. [Barrett, 1979: 75-87] It consists of three steps, which are

- a. Remove all empty transitions.
- b. Remove all transitions to multiple states on one character and replace them with a single transition to a new state. Place the transitions of all the old states into the new state.

c. Remove all inaccessible states.

A series of figures best demonstrates this algorithm. Figure 5 shows a set of regular expressions and the FSA created from them. Figure 6 shows the FSA after removing the empty transitions. Since there are multiple transitions on 'a' from state 1, a new state must be created. Figure 7 shows this conversion. Then, since there are still two transitions from state '2,7,9' on 'b', one more conversion must be performed. Figure 8 is the result. The entire FSA is now deterministic. However, the last step must still be performed. States 2, 3, 4, 6, and 8 may be removed. Figure 9 shows the final DFSA. Examination shows that both the original and the final FSA describe equivalent finite state machines.

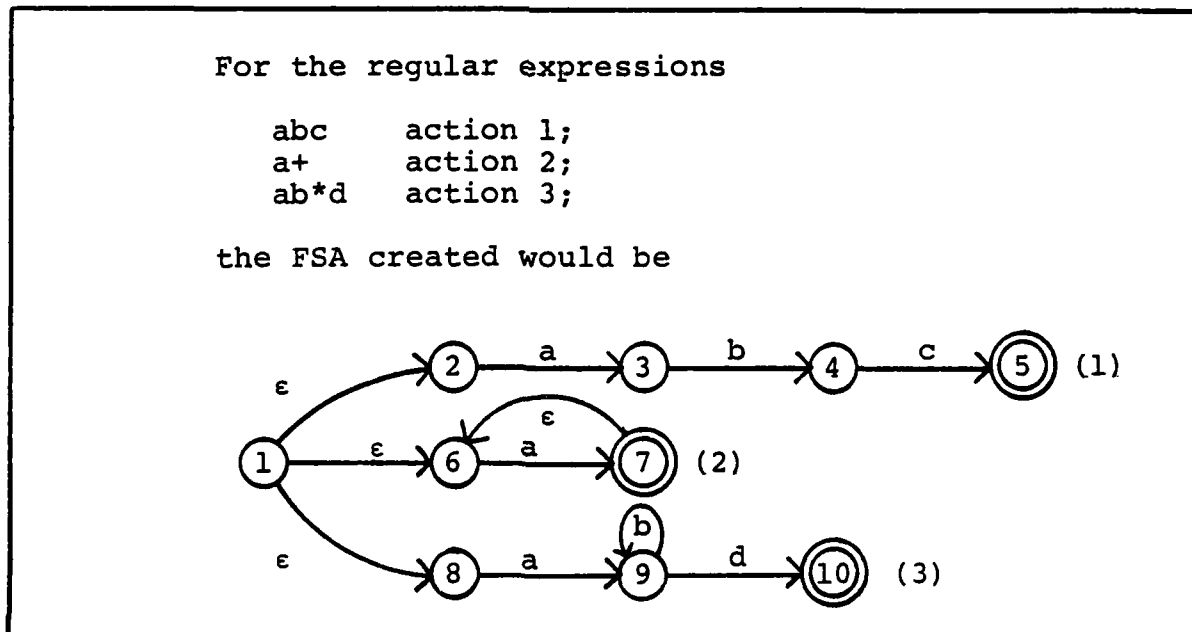


Figure 5. Non-Deterministic FSA Example

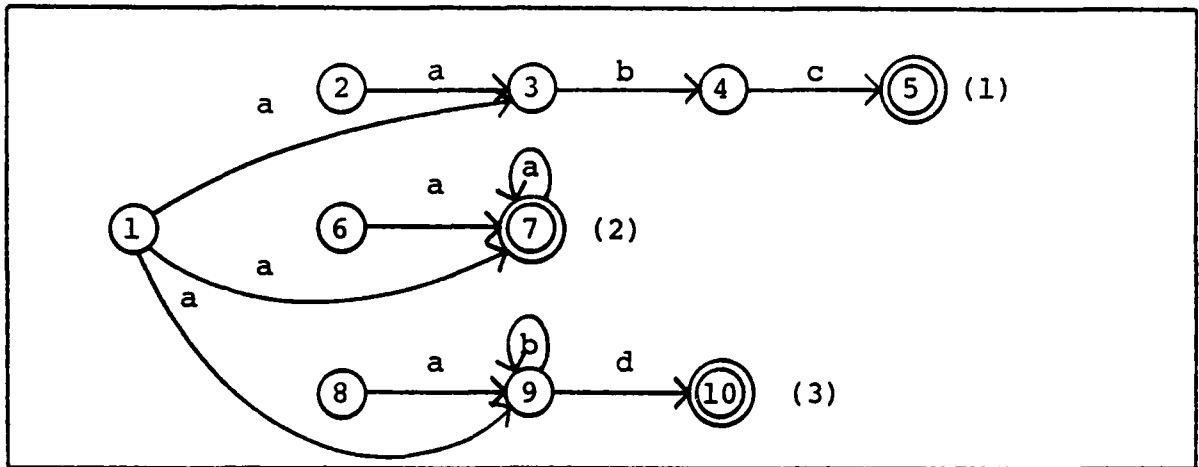


Figure 6. Empty Transition Removal

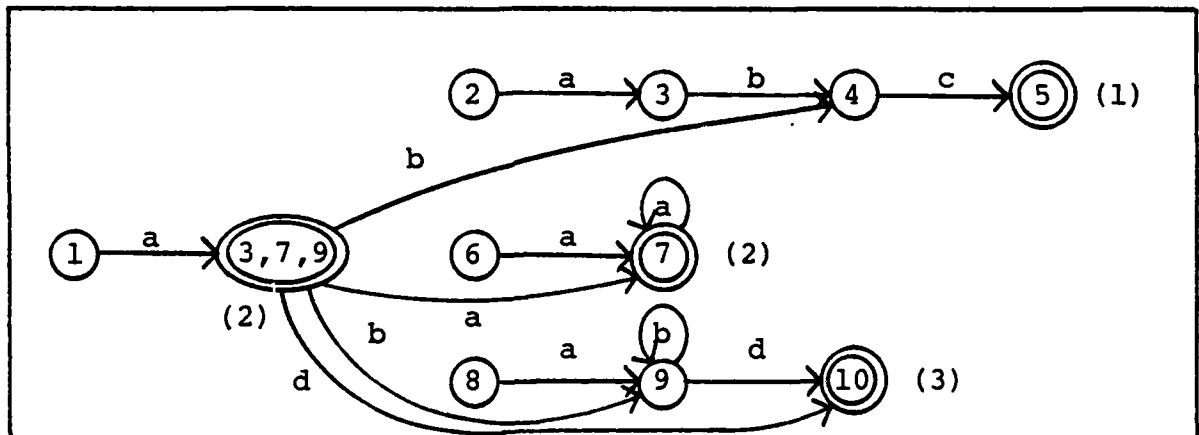


Figure 7. Conversion of Multiple Transitions

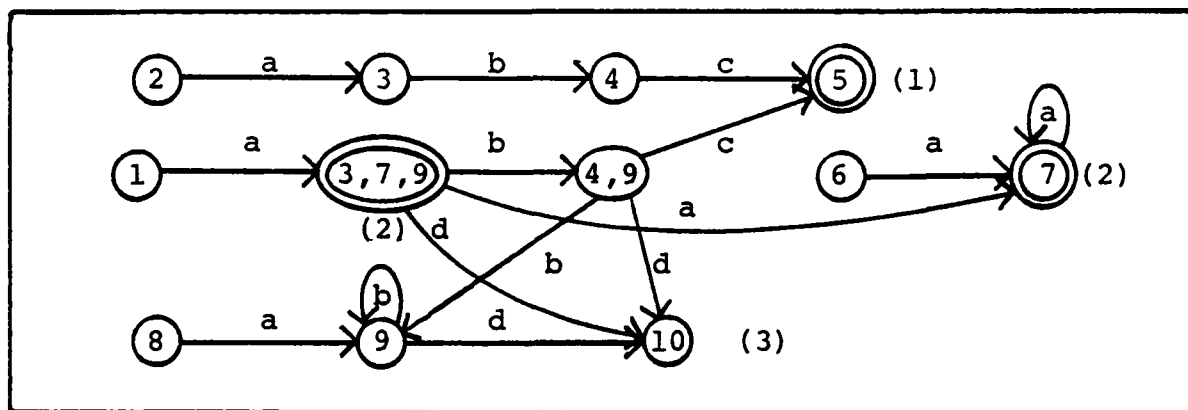


Figure 8. Last Multiple Transition Conversion

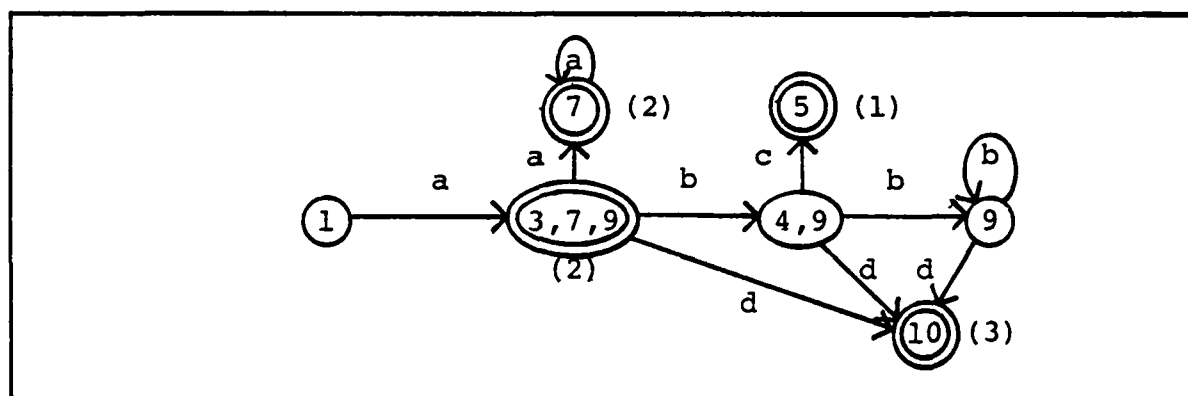


Figure 9. After Deleting Inaccessible Transitions

Once the FSA has been defined, all that remains is to output the appropriate code for each state.

### Decorating the Production Trees

As stated earlier, this particular algorithm was the most difficult to create. The objective is to have the  $\text{first}_k$  sets on the child(ren) of each node for which  $k$  is greater than 0. The first sets of the children are important because it is through their first sets that one can determine

whether or not to take a particular path through the tree.

The algorithm to accomplish this has four basic parts, which are organized to accomplish their task as shown in Figure 10. (The Conclusions chapter gives the practical reason why the algorithm does not merely find the first and follow sets for the maximum possible  $k$ , and avoid the looping construct.)

The algorithm to find the first and follow sets follows the algorithms in Appendix A very closely. Of course, since  $k$  can be greater than 1, some parts of the algorithm were more difficult, but it was a straightforward extension. It was interesting that these two algorithms were the most troublesome. (See the Test Plan chapter.) This probably happened because the first and follow set data structures are manipulated quite a bit, and therefore the number of places for possible error increased greatly.

```
Find maximum possible k          (part 1)
FOR i = 1 to maximum k do
  Find first sets at k = i        (part 2)
  Find follow sets at k = i       (part 3)
  Find k for each node and determine whether
    all nodes have k <= i         (part 4)
  If (all nodes have k <= i)
    THEN stop
  END IF
END FOR
```

Figure 10. Decorate Production Tree Algorithm



Determining  $k$  for each node was a relatively simple task. For each node which could have  $k > 0$ , there is a simple test which determines  $k$ . These tests are:

- a. For closure, closure-plus, and option, compare the first set of the child to the follow set of the node.
- b. For alternation, compare the first sets of the two children.

It should be noted that before determining  $k$ , each node needs to have its first sets adjusted by concatenating the follow sets on to them. This takes care of the situation where the follow set of a particular non-terminal is significant when the non-terminal is  $LL(k)$  but not strong  $LL(k)$ .

Determining Maximum Possible  $k$ . The method for determining the maximum possible  $k$  is relatively straightforward. The idea is to count the absolute largest lookahead that might be possible at the beginning of a parse. Each node has a particular counting method, and these are described in the following list:

- a. empty node : 0
- b. terminal : 1
- c. non-terminal : If this non-terminal is recursive, and it has already been examined, return 1 and a recursion indicator. If not, count the non-terminal.
- d. closure, closure-plus and option: count the child.
- e. concatenation : count each child. If the left child has the recursion indicator set, return its value, otherwise

add the two counts and return.

f. alternation : count each child and return the larger. (A more detailed description and a non-mathematical proof of this algorithm is in Appendix H.)

Final Parser Output. Once all of the decorating has been completed, all that is left is to walk the trees and output the parser. The format of the parser is described in the Parser Section of the next chapter, so the code for each node will not be described here.

## VIII. Parser and Lexical Analyzer

### Introduction

The character of the LL project depended on the output code at least as much, if not more, than it depended on the input. This chapter is devoted to the structure of the output parser and lexical analyzer. It will discuss the general structure, the data structure and code for the lexical analyzer, the data structure and code for the parser, and the method used to remember the values of items in productions.

### General Structure

The general structure of LL output is fairly straightforward. There are seven divisions, which occur in order from the beginning to the end of the output as follows:

- a. the variables and subroutine declarations specified by the user
- b. statements that define each token as an integer for use in all parts of the program
- c. the global variables, to include a definition of the maximum token size, the declarations for the character string of the "current" token during a parser, and the declarations for that token's length
- d. the variables, code, and support subroutines for the lexical analyzer
- e. the variables, code, and support subroutines for the

parser

- f. the variables and subroutines that support remembering the values of items in a production
- g. general support routines.

### The Lexical Analyzer

As seen in the Requirements and Design Decisions Section of Chapter IV, the lexical analyzer must return the integer value of a token recognized, as well as the string that is the token. Also, the lexical analyzer must be a deterministic finite state automaton. To accomplish this, the following data structures, support subroutines, and code were created.

#### Data Structures.

- a. A stack of characters that have been recognized, along with the action number that applied to the state that recognized the character.
- b. A character array that an entire input line will be read into, and characters retrieved from and returned to by the DFSA.

Support Subroutines. The support subroutines all support the reading of data from the input, and the maintaining of that data. All of these manipulations are performed by the subroutines, and the only interface to the lexical analyzer is through two routines that get characters from the input, and put characters back onto the input.

Code. The lexical analyzer itself consists of one routine. The structure of that routine is in Figure 11.

```

while (forever)
  "Begin a new token"
  set current state to 1
  initialize the stack
  while (there was a transition from the previous state)
    get a character and place it on the stack
    case (current state)
      state 1 : "state code"
      state 2 : "state code"
      :
      state n : "state code"
    end case
  end while
  while (the last state on the stack does not have an
    action) pop the stack and put character back on the
    input
  end while
  case (action number on top of stack)
    action 1 : "action code supplied by user"
    action 2 : "action code supplied by user"
    :
    action m : "action code supplied by user"
  end case
end while

```

Figure 11. Structure of the Lexical Analyzer

The "state code" is what is derived from the DFSA data structure. For each character that has a transition from that state, there will be an IF statement. For example, if a state has transitions on 'a', 'b', and 'c' to states 1, 2, and 3 respectively, the state code would be

```

Place state's action number on stack
IF      (current char = 'a')
THEN current state = 1
ELSE IF (current char = 'b')
THEN current state = 2
ELSE IF (current char = 'c')
THEN current state = 3
ELSE there is no transition from this state
END

```

The action code is taken directly from the input. The while loop that surrounds the entire subroutine exists in case a "return" does not appear in an action, so that some sequence of characters is not passed back to the parser as a token, and the lexical analyzer must find the next token.

### The Parser

The parser consists of many subroutines. The technique is to create a single entry point for the user to call which begins the parse, and then make each non-terminal and each action be a separate subroutine. The entry point's only function is to call the subroutine for the start symbol, and to test whether the parse read all the input.

Inside each non-terminal subroutine, the code consists of groups of standard pieces. These pieces are generated by walking the production trees, and the code generated for each type of node (without the value manipulation code) is shown in the following list:

```
Terminal      :  get a token
                  if (token not = terminal-id)
                    then syntax error
                  end if

non-terminal   :  call the non-terminal subroutine

concatenation :  "code for left child"
                  "code for right child"

alternation    :  if (next k tokens equals a first set
                    of the left child)
                    then "code for left child"
                    else "code for right child"
                  end if
```

```

closure      : while (forever)
                if (next k tokens equals a first set
                  of the child)
                  then "code for child"
                  else break out of while loop
                end if
            end while

closure plus  : while (forever)
                "code for child"
                if (next k tokens equals a first set
                  of the child)
                  then do nothing
                  else break out of while loop
                end if
            end while

option       : if (next k tokens equals a first set
                  of the child)
                then "code for child"
            end if

empty        : call the action subroutine
                { if there is an action }

```

There are two major subroutines needed for these routines. The first is the "get a token" subroutine. Its purpose is to accept a token, and remove it from the input. The second is the "lookahead" subroutine. This is used during the various IF statements to compare the first sets against what is in the input. To perform these functions, a table of size k is available, which will contain tokens that have been accepted from the lexical analyzer. Whenever lookahead is requested, enough tokens will be requested from the lexical analyzer to do the compare, and whenever a token is retrieved by the "get a token" routine, one entry will be removed from the table.

## The Values of Productions

The most important part of the generated parser is its ability to place a value on each item in a production. This allows the user to "remember" how many, or what kind, or any other information desired. These values are all integers, therefore any type of information that fits into an integer can be used. In particular, pointers may be used. The following example shows how to use these values. (The \$n values are used as variables in actions.)

S :	A	\$1 has the value of production A
	b	\$2 has the value of terminal b
	={...}	\$3 has the value 0, unless the action used \$\$ = ...; in which case \$3 has that value. This action may reference \$1 and \$2.
	( a b )	\$4 (see note)
	{ a b }	\$5 (see note)
	[ a b ]	\$6 (see note)
	{ a b }+	\$7 (see note)
	={...}	\$8 same as \$3, except that \$1 through \$7 may be referenced.

The value of S is whatever the last item's value was, in this case, whatever \$8 is.

Note: Any group of items surrounded by (), [], {}, or {}+ is a set on its own. So, each of \$4 - \$7 has a subset, and in each one, a = \$1 and b = \$2. \$4's value is whatever b's value is. However, each of \$5 - \$7 may have an unknown number of occurrences. Therefore, each one is represented as an array. The 0th element has the number of occurrences, and the 1st through the nth are the values. So, for { a b }, with input ababab, \$5[0] = 3, and \$5[1] = \$5[2] = \$5[3] = b. This is valuable if the last item is an action, so that the various values may differ. If the last item in a set is one of these multiple occurrences, then the set has the value of the last occurrence.

The values are placed on two large stacks. One stack is the "normal" stack, and contains the values of all items in a



set. If any item in a set is a closure of some type, then the value will point into the second stack, where each element of a closure will be held, be there 1 or 1000.

To go along with these stacks, there are manipulation routines. These include:

- a. Begin a new set
- b. End a set
- c. Create a new entry within a set
- d. Check the stacks for overflow

The "begin a new set" routine returns a pointer to the new set, which is placed in local variables in the parsing sub-routines. This data is used in actions to retrieve the various \$ values. (A limitation on the actions is that only integers and identifiers may exist between the brackets of a \$n array reference.)

Additional insight into the uses of this value capability can be found in the YACC reference manual, [Lesk, 1976] from which LL's was patterned.

## IX. Test Plan

### Introduction

An important facet of any good design is a test plan. Many times this test plan is extremely formal, with benchmarks, acceptance criteria, etc. The amount, complexity and completeness of the test plan is dependent on a number of variables. First, is it possible to test every aspect of a program? The author has worked on projects that were online systems, and there was some code that could only be "assumed" correct until an error occurred. Next, how important is the program? Code that performs missile targeting is obviously critical, while code to calculate a batting average might not receive as much attention if it fails. Finally, how often will the program be used? A job scheduling algorithm in an operating system is used constantly, but most student projects are used only once.

For a project that is relatively large, such as LL, these same criteria apply to each section of code. There are also other criteria that are important to large projects. The most important is the "tightness" of the design. Well-designed programs that have straightforward data structures, and whose interfaces between modules are very small, can be tested in different ways from "loose" programs. The testing program for LL reflects these considerations. LL was de-

veloped in a modular manner, and can be tested in the same way.

### General Test Plan

Testing for LL was done by major submodule. As can be seen from the Data Flow Diagrams in Appendix C, LL's structure is very linear. This means that data structures are operated on in a subsystem, and only the data structure is passed between subsystems. Also, LL's data structures are extremely consistent and general. Therefore, there are very few special cases that need to be tested. (Special cases are the bane of programmers. Each special case must be tested separately, and sometimes they can double or triple the testing workload.)

Each subsystem of LL was tested as it was developed. A set of inputs that exercised every general case was developed and used for each subsystem as it was developed. Extra routines were created whose task was to print the data structures, and these were used extensively during the tests of each subsystem. Also, a trace facility was set in place from the very beginning, which can be turned on and off at will. This trace capability aided testing tremendously, as the location of an error could be exactly placed most of the time.

When all the normal tests were made, then the YACC input was modified, and used as a test case. It succeeded, but caused some repercussions in the LL algorithms. (See the Recommendations Section of Chapter X for an explanation.)

### Special Cases

A special case is essentially some input that causes a global effect on the program, and must be carefully considered during some phase(s) of the development. There were three special cases in LL. These were: recursive productions (for first and follow set calculation), closure as the last item in a set (for remembering the value of a subset of a production), and no regular expression input (for users who want to use their own lexical analyzer).

Recursive Productions. Because the algorithm for calculating first and follow sets requires that each non-terminal be calculated when encountered, each non-terminal had to be marked when it was calculated, so that endless loops would not occur. For example, the production

$$S : a S \mid b ; \quad (76)$$

would have the first sets calculated as in Figure 12. If the algorithm calculated the non-terminal S when it saw it without considering recursiveness, it would continue forever.

```
Begin calculate S
  Begin calculate alternation
    Begin calculate concatenation
      Begin calculate a
      End calculate a
      Begin calculate S
      End calculate S
    End calculate concatenation
    Begin calculate b
    End calculate b
  End calculate alternation
End calculate S
```

Figure 12. Example of First Set Calculation

Closure as the Last Item in a Set. As specified in the Values Section of Chapter VIII, each subset of a production has a value. Whenever closure, closure plus, or option is the last item in a set, the value of that set is the last item in the array of closure items. Therefore, an indicator as to whether or not the last item as a closure was needed.

No Regular Expression Input. The FSA for a grammar is "hung" from a global variable. If no regular expressions are found in a grammar specification, then this variable is set to null, and when the final output is created, no lexical analyzer code is output.

#### The ADA Test

As we saw in the introductory chapter, ADA [DARPA, 1981] is an on-going project to create a new high-level language. Since it is very complex, and the justification for work such as this thesis comes from languages such as ADA, the author decided to create a test case using the ADA EBNF specification. [DARPA, 1981: E1-E5] The outputs from LL, along with an explanation of the test, are in Appendix I.

## X. Conclusions and Recommendations

### Conclusion

Throughout the course of this thesis project, there were three main questions. There were:

- a. Can a generalized LL(k) parser generator be created that determines what k is?
- b. Is the parser generated relatively economically?
- c. Is the output product easy to understand?

The answer to question "a" is a qualified yes. It is qualified because of a flaw that could not be rectified within the time constraints. This relates to LL(k) grammars that are not strong LL(k), and is only a problem when the applicable follow sets must traverse more than one production to reach the production that is not strong LL(k). For example, if the productions are

$$S \rightarrow aBbc \mid Bc \quad (77)$$

$$B \rightarrow b \mid c \mid \epsilon \quad (78)$$

then all is well. The two references of B will be used to good effect to separate the follow sets. If, however, the productions are

$$S \rightarrow aBbc \mid Bc \quad (79)$$

$$B \rightarrow C \quad (80)$$

$$C \rightarrow b \mid c \mid \epsilon \quad (81)$$

then there is only one reference of C, and the multiple refer-

ences to B (with their follow sets) are lost because of the indirect reference.

The second question, is the process economical, is an unqualified no. As built, the data structures are too large and use too much memory for any "reasonably" sized grammar with k larger than two or three. It is, however, excellent for grammars that are relatively small.

The last question is an unqualified yes. The output product is very easy to understand and use. It seems to be organized well, with data structures that fit the necessary algorithms very nicely.

#### Recommendations

There are two problems in LL that make it somewhat less than practical for many applications. These are: 1) core usage, and 2) maximum k calculation.

Core Usage. The only area of concern in LL is the amount of core that is used during a large grammar's processing. The first and follow set calculation seems to be the root cause of this problem. During the testing phase, the YACC input was modified so that it could be read into LL, and it was. All the phases worked well, until the first sets were to be calculated. This calculation for the maximum k came to be 39, and at that time the first sets were not calculated in a loop starting at 1. The algorithm attempted to perform this feat for k = 39. LL began to request more and more memory, and

had barely begun the calculation when it finally reached 6 megabytes (MB) and was killed by the system. At that time, the loop was put in, and the process was restarted. It happened that  $k = 2$  for the grammar, and LL went to 6 MB again, but not until the process was almost completed. At that time, the grammar was modified to be LL(1), and then everything went smoothly. The YACC  $\rightarrow$  LL input modification test showed this failing of LL very well.

Since LL is a prototype, the author decided that simplicity of the data structures and ease of programming, (along with an immovable time constraint), was more important than algorithm efficiency. So, LL constant use of the memory allocate and deallocate functions in UNIX to build the first and follow sets. Each first and follow set item requires one piece of memory, which contains the actual data, along with a pointer to the "next" item. Unfortunately, the memory manager also places a header on each piece of memory that is given to the user. If the data structure was modified so that the memory manager would only be needed occasionally, and so that the "next" pointer could be disposed of, a 75% reduction in core use would occur. (See figure 13(a).) Then, if the data could be compressed so that four data items would fit where only one fits now, another 75% reduction would occur. (See figure 13(b).) If both could be implemented, there would be a 93.75% total reduction in core use, and LL could be used on a much larger class of grammars.



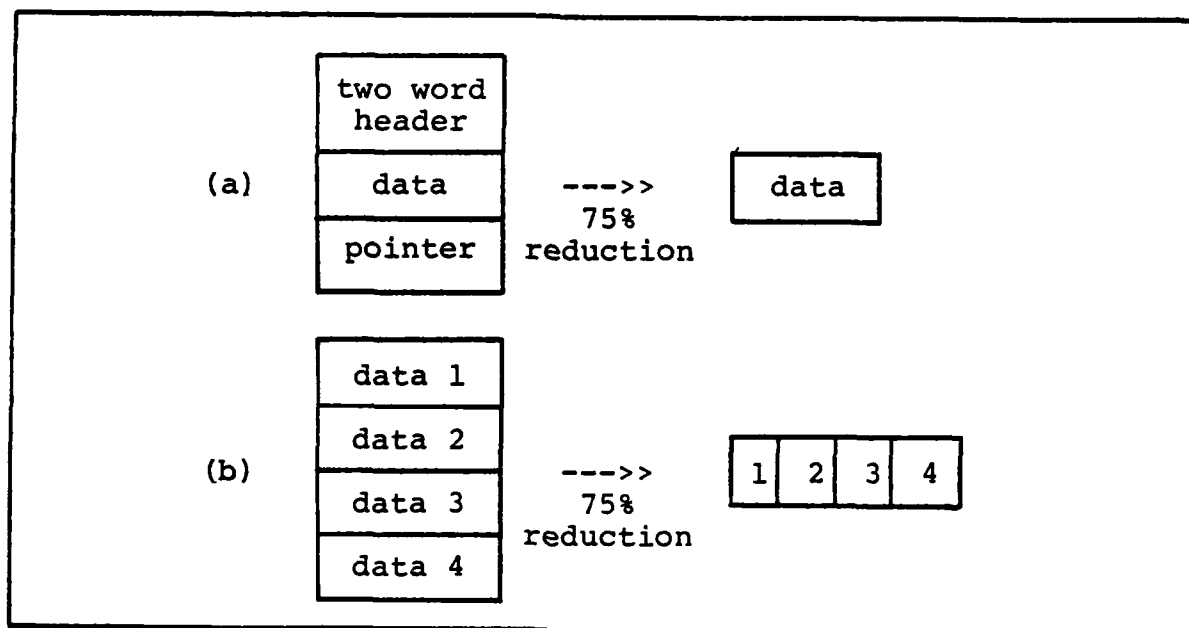


Figure 13. Possible Core Use Reductions

Maximum k Calculation. The other area of concern in LL is in the calculation of the maximum possible k for a grammar. Because this value is LL's limit in its loop to find the actual k, it is advantageous to have it be as small as possible. Unfortunately, for any large grammar, the number calculated by the algorithm in Appendix H is relatively large. This limiting algorithm needs more work to bring the value down.

In order to solve this problem of practicality, an option to the program was added to allow the user to insert a value for the maximum possible k. Thus, if a grammar is known to have k less than or equal to some small value, then this value can be inserted as an option. However, in that case LL can not assure the user that the grammar is not LL(k) for any k,

but only for  $k$  up to the input value.

Final Comments. LL is a prototype that shows that calculation of  $k$  is possible for  $k > 1$  in a language with an LL( $k$ ) grammar. With the above mentioned modifications in place, and possibly rewritten to remove other inefficiencies, there should be no difficulty in using it as a production quality compiler writing tool.

## Bibliography

- Aho, Alfred and Jeffrey Ullman. The Theory of Parsing, Translation, and Compiling (Volumes I and II). Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1972.
- Baker, Theodore P. "Extending Lookahead for LR Parsers". Journal of Computer and System Sciences, 22: 243-259, 1981.
- Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Chicago: Science Research Associates, 1979.
- Berry, R. E. Programming Language Translation. New York: John Wiley and Sons, 1982.
- DARPA (Defense Advanced Research Projects Agency). ADA Reference Manual. US Government, 1981.
- Davie, A. and R. Morrison. Recursive Descent Compiling. New York: John Wiley and Sons, Inc., 1981.
- Gert, Florjin and Rolf Geert. PGEN - A General Purpose Parser Generator. Amsterdam, Netherlands: Mathematical Centre, January 1981 (N81 - 30844).
- Gries, David. Compiler Construction for Digital Computers. New York: John Wiley and Sons, Inc., 1971.
- Johnson, Stephen S. YACC: Yet Another Compiler Compiler. UNIX Programmer's Manual, 7th Ed, Volume 2A, 1978.
- Koster, C. H. A. A Compiler Compiler. Amsterdam, Netherlands: Mathematisch Centrum Amsterdam, November 1971 (AD - 900731).
- Lesk, M. E. Lex - A Lexical Analyzer Generator. UNIX Programmer's Manual, 7th Ed, Volume 2A, 1978.
- Leverett, Bruce W. et al. "An Overview of the Production Quality Compiler-Compiler Project". Computer Magazine: 38-49, August 1980.
- Meertens, L. G. and Vliet, J. C. van. On Top-Down Parsing of Algol 68+. Amsterdam, Netherlands: Mathematisch Centrum Amsterdam, November 1981 (AD - B064127).

Peters, Lawrence J. Software Design: Methods and Techniques. New York: Yourdan Press, 1981.

Pyster, Arthur. Compiler Design and Construction. New York: Van Nostrand Reinhold Company, 1980.

Wirth, Nicklaus. Algorithms + Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1975.

## Appendix A

### FIRST<sub>1</sub> and FOLLOW<sub>1</sub> Set Calculation

This algorithm operates on production trees as described in the Production Tree Section of Chapter II. The only non-obvious part of the algorithm is the symbol  $\perp$ . This symbol stands for the end of input, end of string, or end of file. (Note that the FOLLOW set of the start symbol should include the end of string, or  $\perp$ .) [Barrett, 1979: 180-184]

Set all FIRST sets for all nodes in all trees to the empty set.  
REPEAT

FOR all trees DO  
CALCULATE-FIRST-SET(root of tree)  
END FOR

UNTIL no change in any FIRST set

Set all FOLLOW sets for all nodes in all trees to the empty set.  
Set FOLLOW(root node of start symbol's tree) to  $\perp$   
REPEAT

FOR all trees DO  
CALCULATE-FOLLOW-SET(root of tree)  
END FOR

UNTIL no change in any FOLLOW set

CALCULATE-FIRST-SET(tree node)

CASE node OF

$\epsilon$  : add  $\{\epsilon\}$  to FIRST(node)

terminal x : add  $\{x\}$  to FIRST(node)

non-terminal N : add FIRST(root node of tree N) to  
FIRST(node)

• : CALCULATE-FIRST-SET(left child)

CALCULATE-FIRST-SET(right child)

add FIRST(FIRST(left child) FIRST(right child)) to  
FIRST(node)

?, \* : CALCULATE-FIRST-SET(child)

add  $\{\epsilon\}$  to FIRST(node)

add FIRST(child) to FIRST(node)

| : CALCULATE-FIRST-SET(left child)

CALCULATE-FIRST-SET(right child)

add FIRST(left child) to FIRST(node)

add FIRST(right child) to FIRST(node)

END CASE

RETURN

END CALCULATE-FIRST-SET

CALCULATE-FOLLOW-SET(tree node)

CASE node OF

terminal  $x, \epsilon$  : (\*no action\*)

non-terminal N : add FOLLOW(node) to FOLLOW(root of N)

• : add FIRST(FIRST(right child) FOLLOW(node)) to  
FOLLOW(left child)

add FOLLOW(node) to FOLLOW(right child)

CALCULATE-FOLLOW-SET(left child)

CALCULATE-FOLLOW-SET(right child)

| : add FOLLOW(node) to FOLLOW(right child)

add FOLLOW(node) to FOLLOW(left child)

CALCULATE-FOLLOW-SET(left child)

CALCULATE-FOLLOW-SET(right child)

? : add FOLLOW(node) to FOLLOW(child)

CALCULATE-FOLLOW-SET(child)

\* : ADD FIRST(FIRST(child) FOLLOW(node)) to  
FOLLOW(child)

CALCULATE-FOLLOW-SET(child)

END CASE

RETURN

END CALCULATE-FOLLOW-SET

## Appendix B

### LL(1) Parser from Production Trees

This Appendix gives an algorithm to generate a parser for LL(1) grammars by using production trees that have been decorated with FIRST and FOLLOW sets as in Appendix A. The algorithm assumes an unambiguous grammar, and performs no error-correction. It will, however, recognize a syntax error. [Barrett, 1979: 185-187]

There are two variables. TOKEN is the value of the current token in the parse, and FLAG is used as a boolean variable to decide whether some part of the parse was successful or not. There is also a function called NEXT. NEXT gets the next token on the input stream and returns its value. It should be noted that the symbol  $\perp$ , as in Appendix A, is the end of input symbol. Also, the FIRST and FOLLOW sets will be taken from the tree and placed in the parser wherever FIRST and FOLLOW appear in the algorithm. The parser will be created in pseudo-PASCAL, and is generated by the algorithm that follows.

```
Generate: Program header with declaractions of FLAG and TOKEN.
FOR each tree DO
  Generate:
    "procedure A; " where A=non-terminal symbol for the tree
    "begin"
    GEN-CODE(tree root)
  Generate:
    "end"
END FOR
Generate:
```

```

"begin"
  "call G;" where G is the start symbol
  "if (FLAG = FALSE) or (TOKEN <> '␣') then ERROR;"
"end"

GEN-CODE(tree node)
CASE node OF
  ε : Generate:
    "FLAG := TOKEN in FOLLOW(node)"
  terminal x : Generate:
    "FLAG := (TOKEN in 'x');"
    "if FLAG then NEXT;"
  non-terminal N : Generate:
    "call N;" (*proc for non-terminal n*)
  * : Generate:
    "if TOKEN in FIRST(child)"
    "then begin"
    "repeat"
      GEN-CODE(child of node)
    "until not FLAG;"
    "FLAG := TRUE;"
    "end (*begin*);"
    "else FLAG := TOKEN in FOLLOW(node);"
  . : GEN-CODE(left child)
    Generate:
    "if FLAG"
    "then begin"
      GEN-CODE(right child)
    "if not FLAG then ERROR;"
    "end (*begin*);"
  | : GEN-CODE(left child)
    Generate:
    "if not FLAG"
    "then begin"
      GEN-CODE(right child)
    "end (*begin*);"
  ? : GEN-CODE(child)
    Generate:
    "if not FLAG"
    "then FLAG := TOKEN in FOLLOW(node);"
END CASE
END GEN-CODE

```



Appendix C  
Data Flow Diagrams

System Diagram

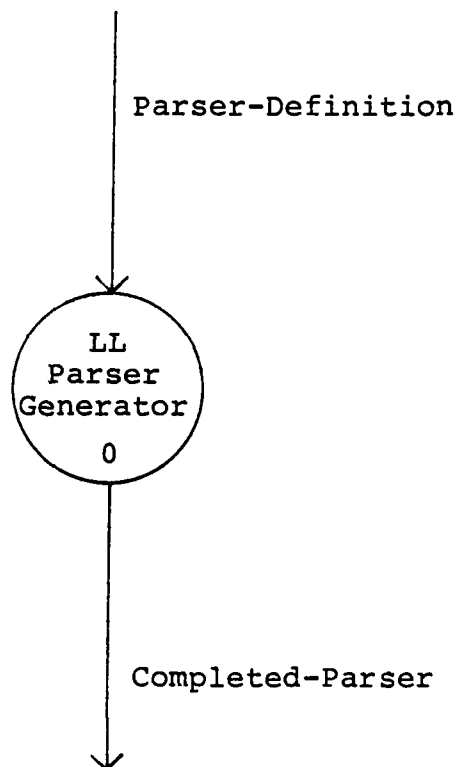


Diagram 0

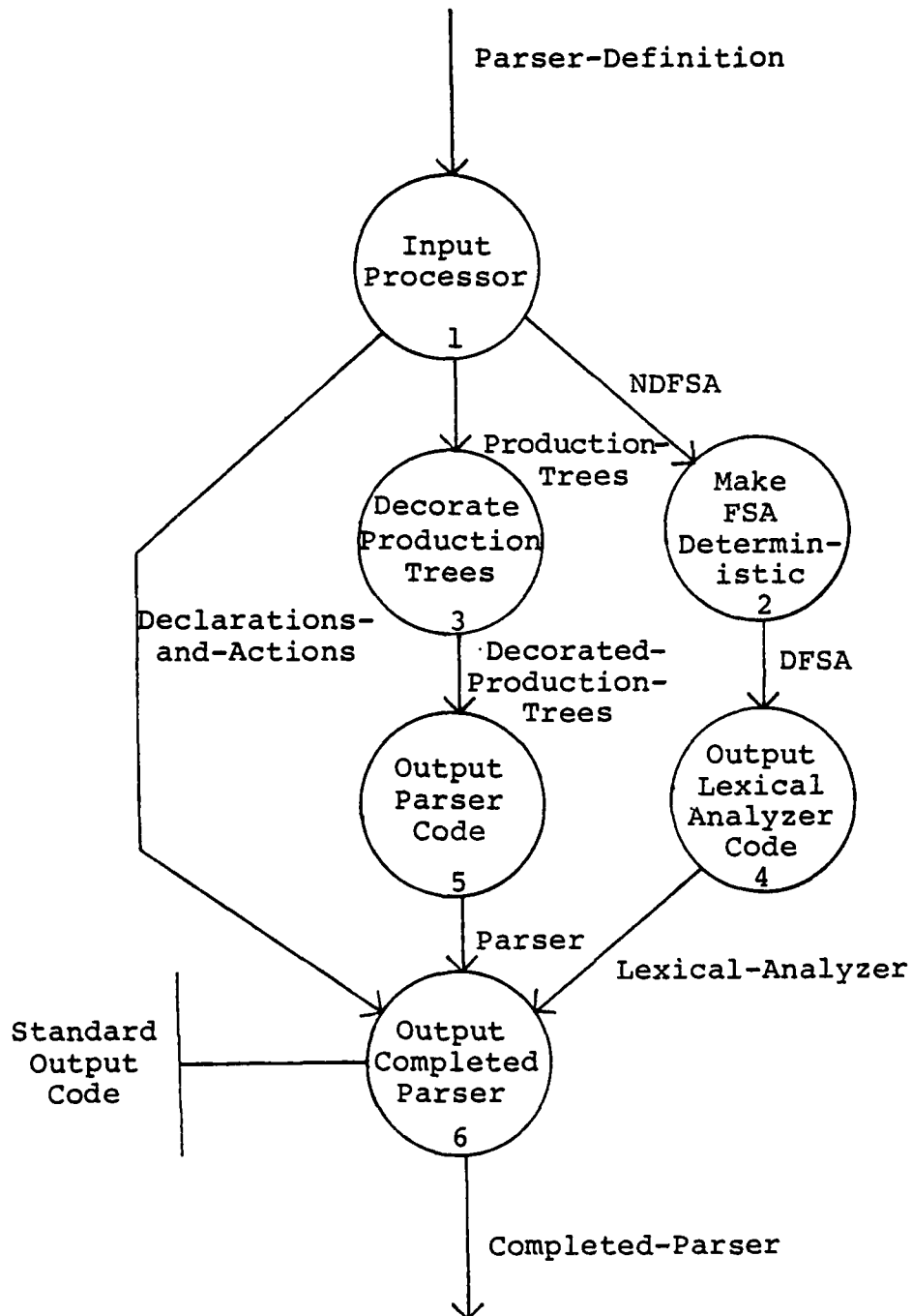


Diagram 1

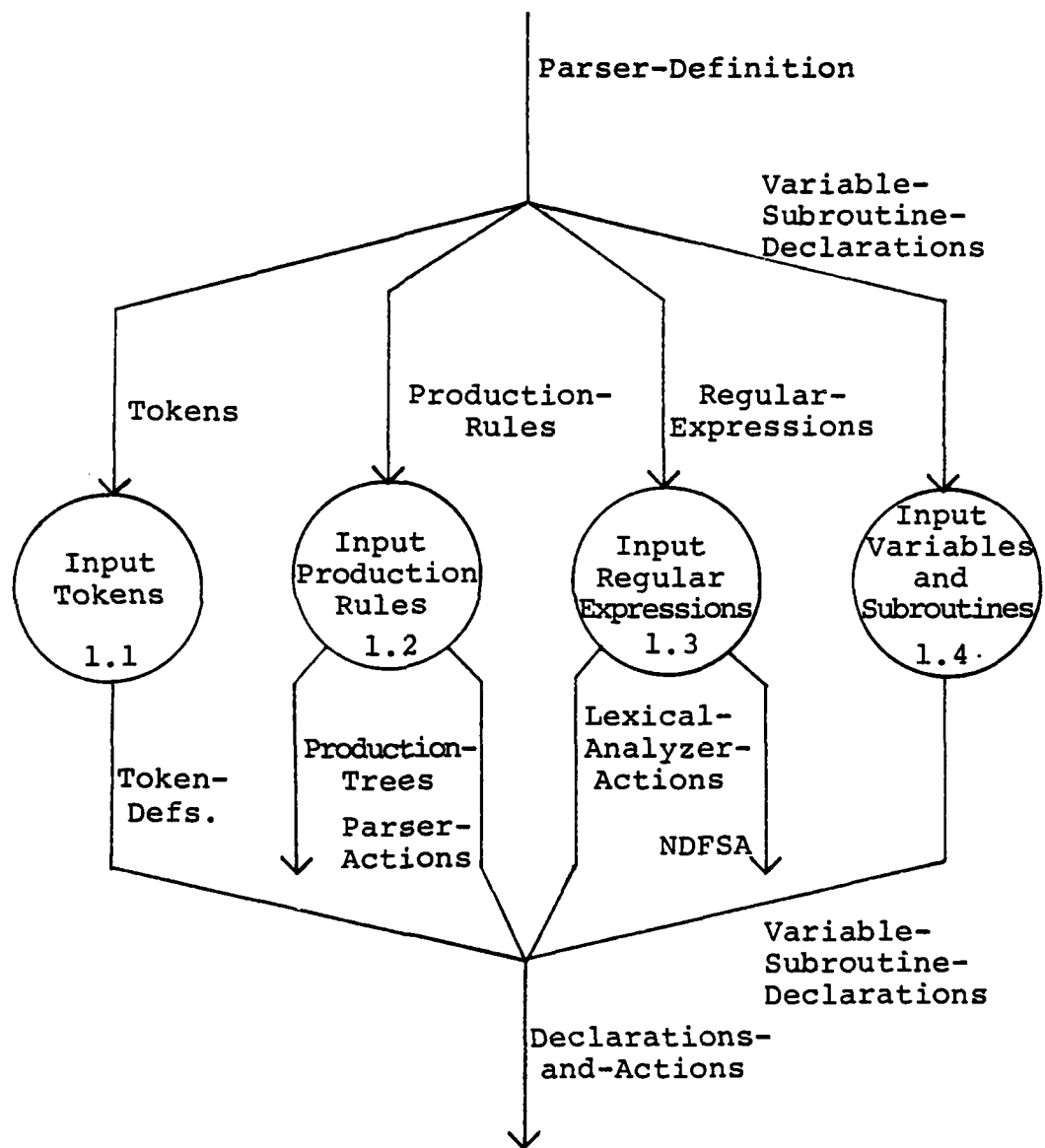


Diagram 2

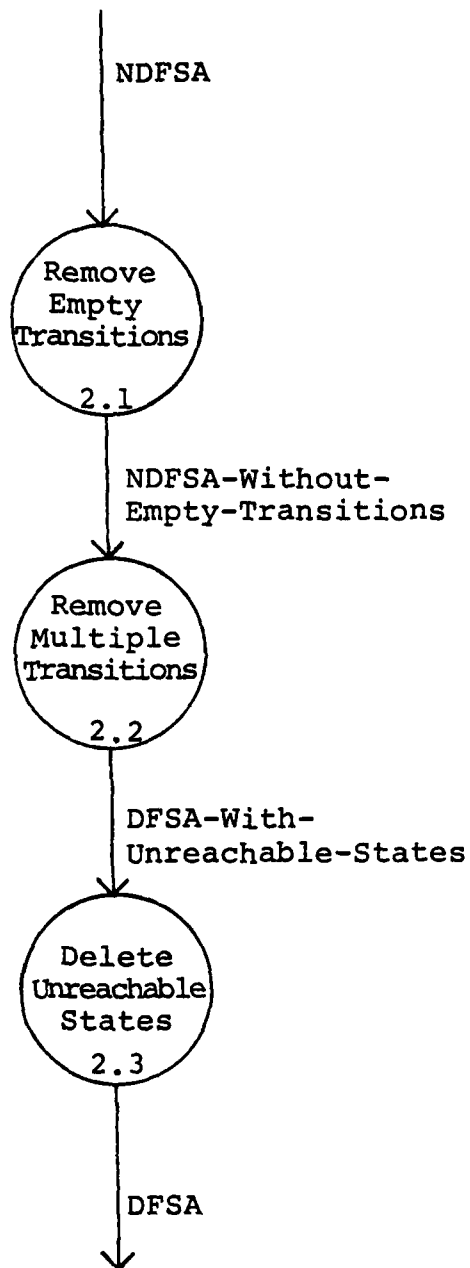
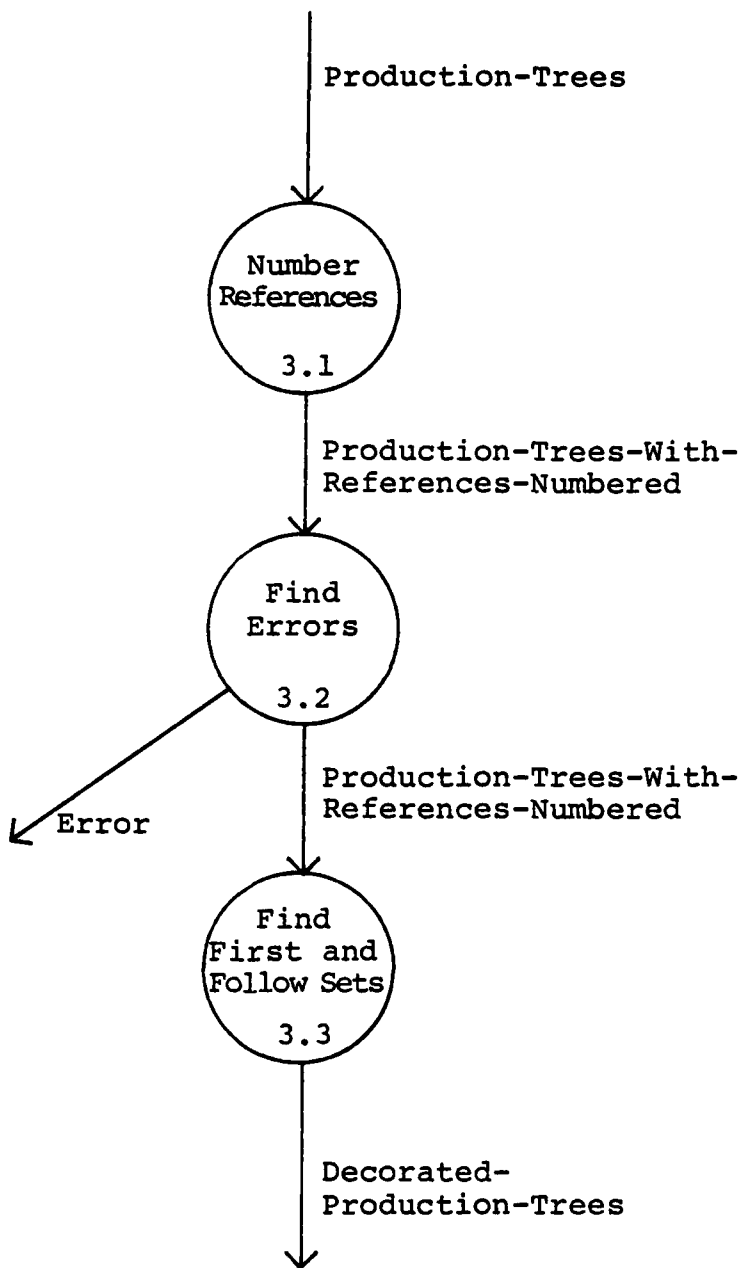
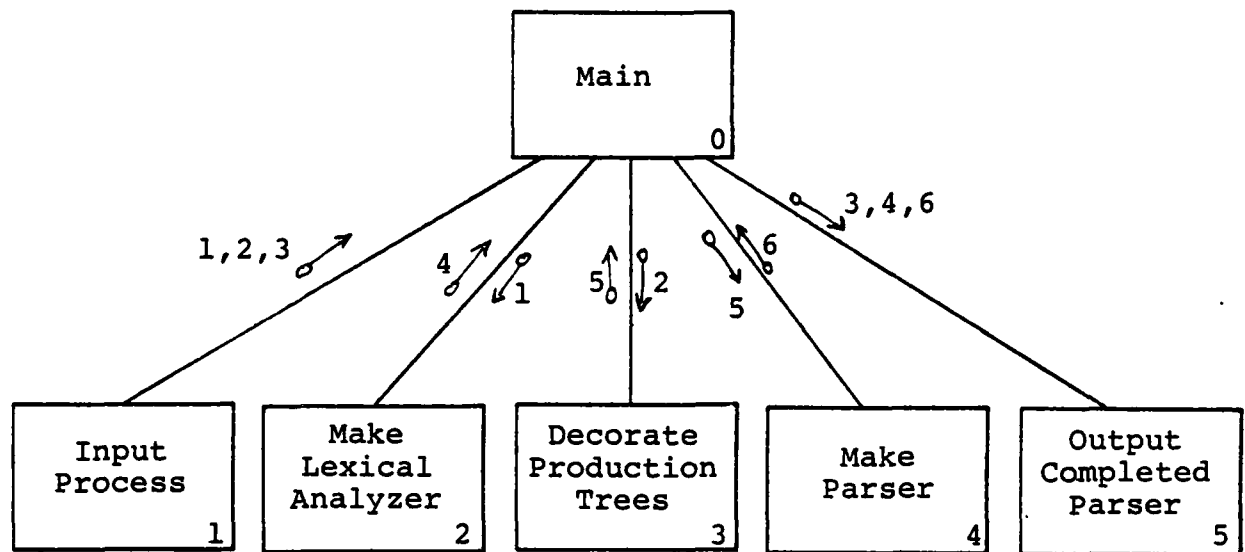
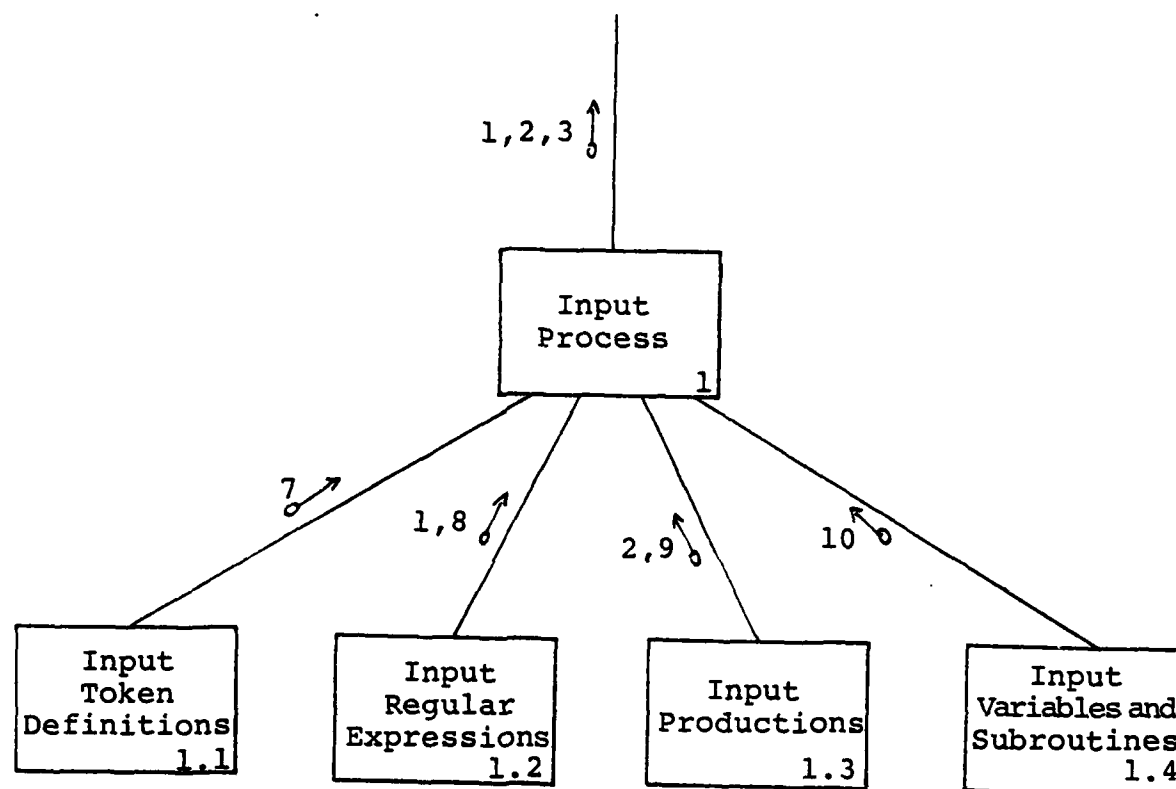


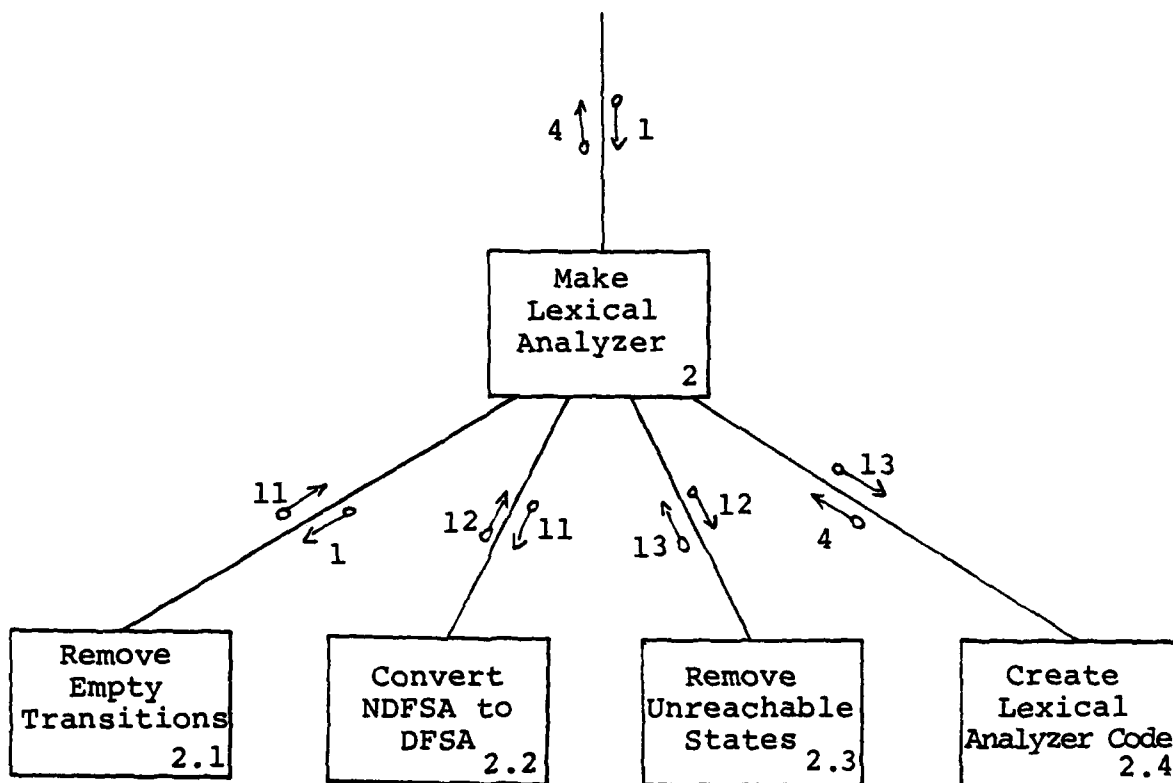
Diagram 3



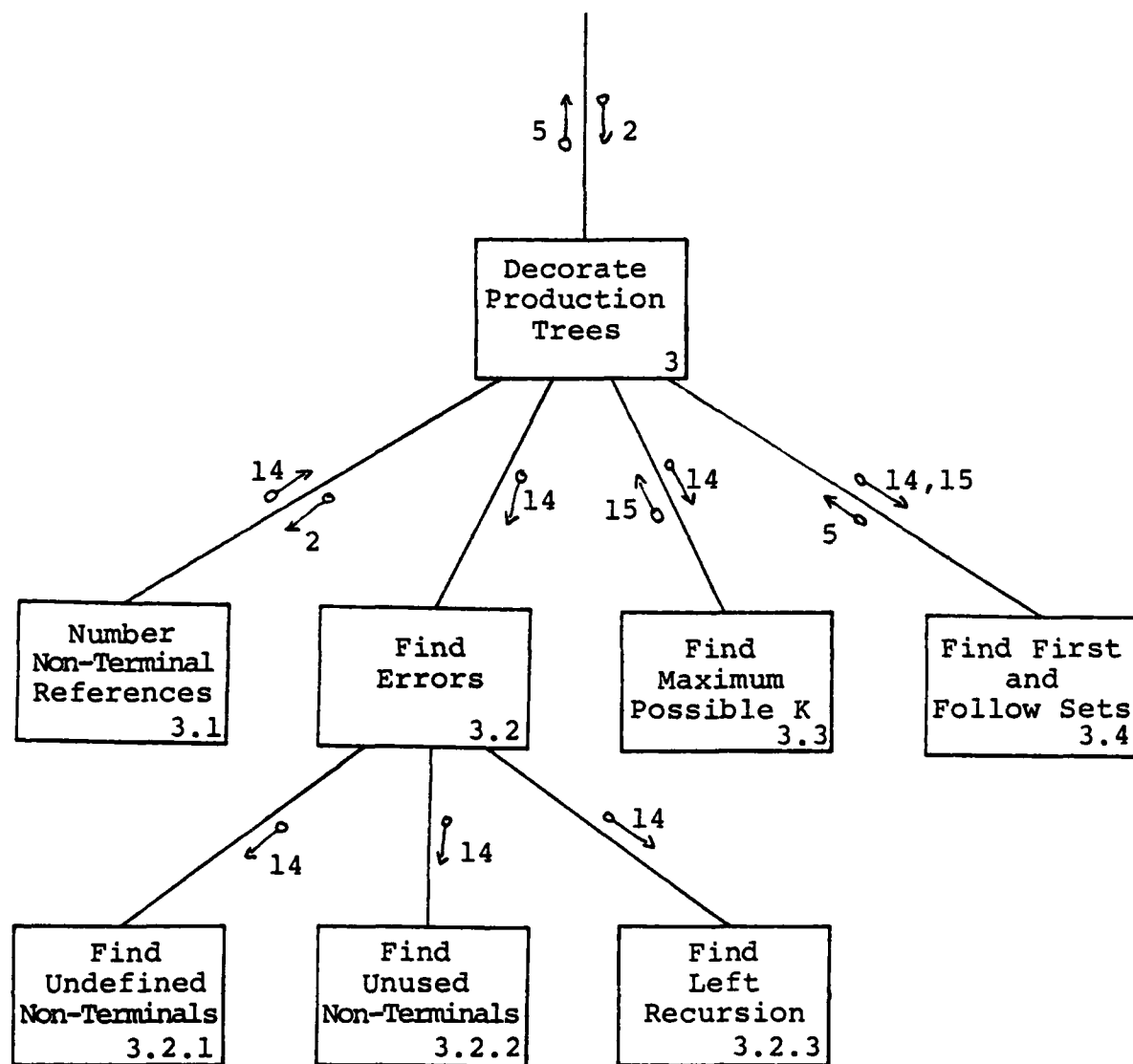
Appendix D  
Structure Charts

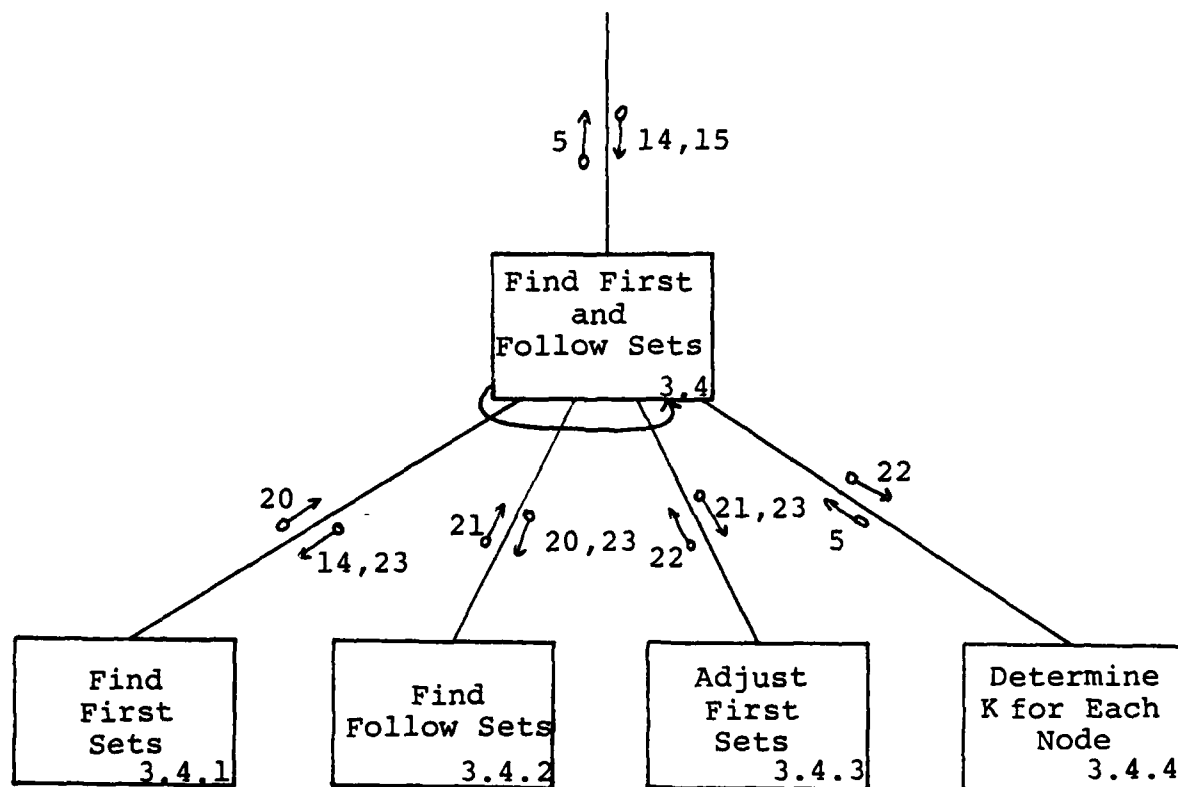


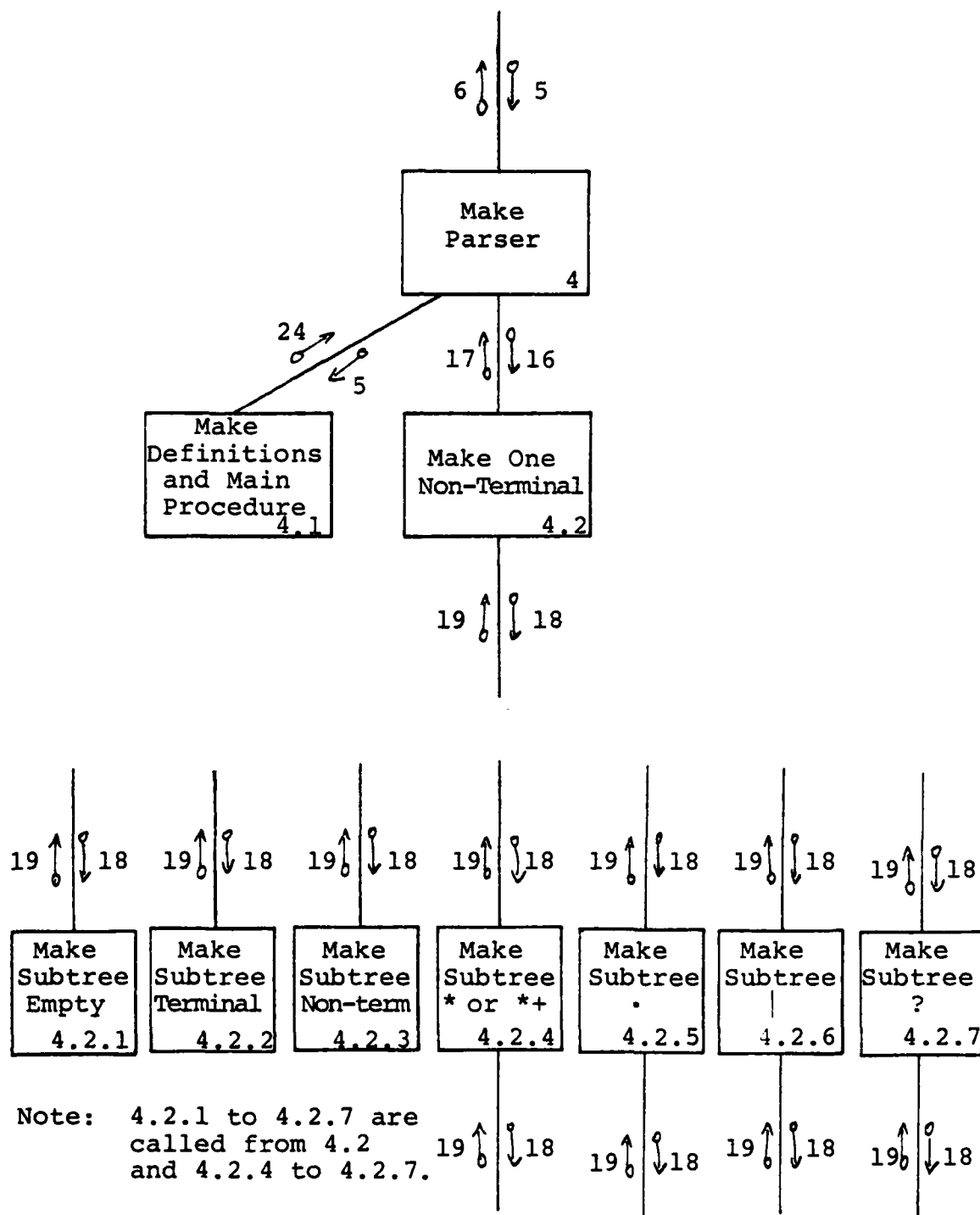












## Numbered Data Items List

1. Non-deterministic FSA
2. Production Trees
3. Complete grammar input. Includes (7) through (10)
4. Lexical analyzer code
5. Decorated production trees
6. Parser code
7. Token definitions
8. Lexical analyzer actions
9. Production actions
10. Variable/subroutine declarations
11. NDFSA without empty transitions
12. DFSA with unreachable states
13. DFSA
14. Production trees with numbered references
15. Maximum possible k
16. One production tree
17. Code for one production tree
18. Subtree
19. Code for subtree
20. Production trees with first sets
21. Production trees with first and follow sets
22. Production trees with first set of length k
23. Temporary value of k
24. Main procedure for output code

## Appendix E

### Regular Expression Forms Used in LL

regular-expression : alternate { "|" alternate } ;

alternate : { element }+ ;

element : sub-element [ "\*" | "+" | "?" ] ;

sub-element : "(" regular-expression ")"

          | "[" [ "^" ] { bracketed-char }+ "]"

          | "

          | sub-element-character

;

bracketed-char :

Any character defined for C, to include octal numbers for unprintable characters, any of \t, \n, etc., excluding ']'. Any character, including \ and ], may be preceded by a \ to force it into the character set.

quoted-char :

Same as for bracketed-char, except that " is the character that may not be included in the set unless it is preceded by a \.

sub-element-char :

Any character defined in C except for space, newline, tab, [ and ", unless preceded by a \.

Appendix F  
Production Rule Format

```
production-rule : non-terminal-IDENTIFIER
                  ":"
                  alternate-list
                  ";"
                ;

alternate-list : alternate { "|" alternate } ;
alternate : { item } ;
item : "(" alternate-list ")"
      | "[" alternate-list "]"
      | "{" alternate-list "}" [ "+" ]
      | IDENTIFIER
      | "{ action-code " }"
      ;
```

AD-A138 061

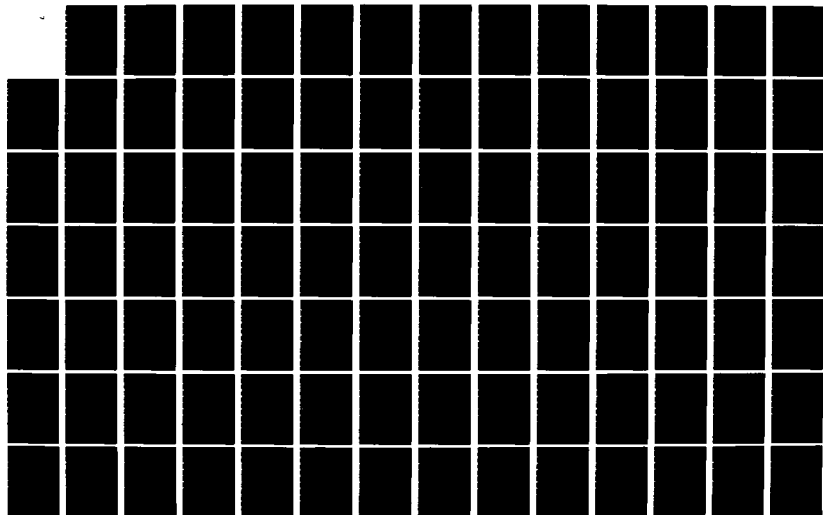
A GENERATOR OF RECURSIVE DESCENT PARSERS FOR LL(K)  
LANGUAGES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING G B PAPROTYN DEC 83  
AFIT/GC5/MA/83D-6

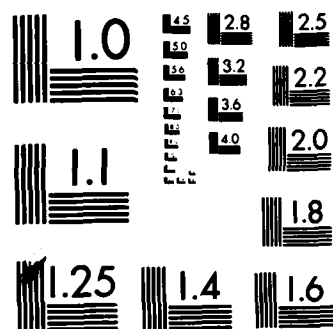
2/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



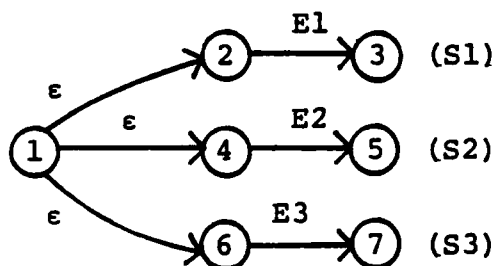
## Appendix G

### FSA for Each Regular Expression Form

In the following structures,  $n$  denotes a state, and  $n$  denotes a stop state. The beginning state is always state 1, and the ending state is marked with an S. An expression may be any of the following, or (E).

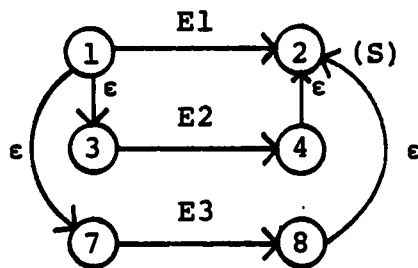
---

E1    action 1  
E2    action 2  
E3    action 3

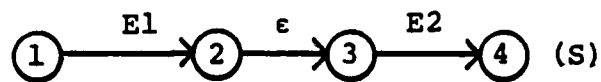


---

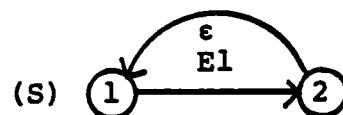
E1 | E2 | E3



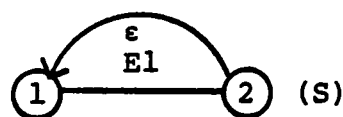
E1 E2



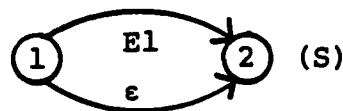
E1\*



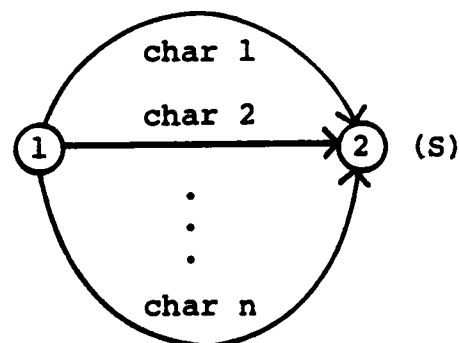
E1+



E1?



[ set of chars ]



## Appendix H

### Maximum Possible K Algorithm

This appendix describes and justifies the algorithm for calculating the maximum possible k. The author has developed this algorithm through his knowledge of what lookahead is possible for each of the nodes, and then had tested on a number of cases that used differing combinations of nodes.

The algorithm begins at the head node for the start symbol, and essentially traverses all the production trees during its calculation. The following list describes the algorithm for each production tree node in turn, and gives the justification of the algorithm for that node.

#### General Comments

- a. Counting is the process of determining what the maximum possible lookahead is for a node.
- b. The recursion indicator is set in order to remember whether the counting process for a particular non-terminal node ended when recursion was encountered.
- c. The maximum k value is a variable that is set whenever a node is found that has a value b greater than the value that is in the variable. This is done because the maximum value for the start symbol may be smaller than some subordinate production's value.

### Empty Node

This node is counted as 0, because nothing will be parsed for it.

### Terminal Node

The terminals are all counted as 1, because exactly one token will be parsed.

### Closure Node

A closure node defines a sentence that may occur 0 or more times. The problem in determining the maximum k for this node is: How many times will the sentence occur in the input, and how many occurrences of the sentence must be looked ahead to determine whether to accept it or not. There are two separate considerations here, which are:

a. When parsing the actual closure node, the maximum k is the count of the child, since that is the maximum that can be viewed at that point.

b. If the closure is inside an alternation node, do we parse the closure node or not? This problem is more complex. The following examples show various occurrences of closure nodes in grammars:

$$S \rightarrow \{ a b \} c \mid a b a b d \quad (82)$$

$$S \rightarrow \{ a b \} c \mid \{ a b c \} d \quad (83)$$

$$S \rightarrow \{ a b \} c \mid \{ a b \} d \quad (84)$$

Equation 82 is LL(5). This is because the two alternates can parse "ababc" and "ababd". The concatenation node algo-

rithm will return a count of 5, so the closure node's sentence may be counted only once, for a total count on the left hand side of 3. Equation 83 is LL(4), because the alternates may parse "abc|" and "abcd". Once again, each closure's sentence only needs to be counted once. The last example, equation 84, is not LL. This type of closure and alternation problem is the only closure case where the lookahead is unknown. LL will not accept grammars with such productions.

Intuitively, we must count a closure node's child once and only once. If we had to count it twice because of some case, why wouldn't there be a case where it needed counting three times? Or four? The algorithm would break down and be unworkable in this case. Fortunately, such a case does not exist.

#### Closure-Plus

For this node, count the child and return its value, by the same reasoning as for the closure node.

#### Option Node

Since an option node is parsed 0 or 1 times, the maximum possible k is the child's count.

#### Non-Terminal Node

Normally, the maximum k for a non-terminal node is calculated by going to the non-terminal's production and counting it. However, a non-terminal may be recursive, as in

$$S \rightarrow a b S c \mid a b \quad (85)$$

If the above algorithm were used, it would never terminate on this example. The same type of intuitive reasoning applies to this node as applied to the closure node. We should end the counting process when the non-terminal comes up for counting the second time, and return 1 as its count. Thus, the count for the left hand side of the example would be 3. Also, the recursion indicator needs to be set, because the concatenation node will use that to determine whether to use the subsentence to the right of a non-terminal or not. It should not in the recursive case, because if the grammar requires the compiler to look ahead further than the non-terminal, it would not be LL anyway.

#### Concatenation Node

The concatenation node's count is normally the sum of its two children. However, if the left child has the recursion indicator set, then the sum is no longer appropriate, as we saw for the non-terminal node. In that case, the count for the left hand side only will be returned.

#### Alternation Node

Since the alternation node defines the "choose A or B" construction, the maximum possible k is the larger of the counts of its children.

## Appendix I

### LL Output for a Partial ADA Grammar

This appendix contains the grammar input file and the output files for a partial ADA grammar processed by LL. Only a small part of the actual ADA specification was used because the actual ADA grammar is quite large.

The files in the appendix are:

- a. The input grammar.
- b. The C program that LL created.
- c. The LL.output file. This file contains a summary of the FSA for the lexical analyzer, a list of those productions which were not LL(1), and a graphic representation of the decorated production trees. (This file would have a list of all those productions which were not LL(i) for all i from 1 to k. The ADA grammar used here is LL(2).)

# Ada Grammar Subset

declaration expression sequence\_of\_statements operator\_symbol constraint  
 identifier discrete\_range parameter\_association exception\_handler  
 designator

COLON COLON\_EQUALS LEFT\_PAREN RIGHT\_PAREN COMMA ENTRY SEMICOLON  
 IN OUT DOT ALL BEGIN PACKAGE BODY IS EXCEPTION END PRIVATE  
 PROCEDURE FUNCTION RETURN USE WITH

```

xx
"!"
[a-zA-Z][a-zA-Z]*
[0-9]+
"PACKAGE"
xx
return(COLON);
return(identifier);
return(expression);
return(PACKAGE);

```

```

compilation :
( compilation_unit ) ;

```

```

body :
subprogram_body
| package_body
;

```

```

compilation_unit :
context_specification
(
subprogram_declaration_body
| package_declaration
| package_body
) ;

```

```

context_specification :
( with_clause ( use_clause ) ) ;

```

```

declarative_item :
declaration | use_clause ;

```

```

declarative_part :
( declarative_item ) ( program_component ) ;

```

```

formal_part :
LEFT_PAREN parameter_declaration
{ SEMICOLON parameter_declaration }
RIGHT_PAREN ;

```

```

identifier_list :
identifier ( COMMA identifier ) ;

```

```

mode :
[ IN ] | OUT | IN OUT ;

```

```

name :
( identifier | operator_symbol ) name_1 ;

```

```

name_1 :
(
LEFT_PAREN expression ( COMMA expression ) RIGHT_PAREN
| LEFT_PAREN discrete_range RIGHT_PAREN
| DOT identifier
| DOT ALL
| DOT operator_symbol
| LEFT_PAREN [ parameter_association ( COMMA

```



parameter\_association ) ] RIGHT\_PAREN

name\_1  
;  
;

package\_body :  
PACKAGE BODY Identifier IS declarative\_part  
[ BEGIN sequence\_of\_statements  
[ EXCEPTION ( exception\_handler ) ] ]  
END [ Identifier ] SEMICOLON ;

package\_declaration :  
package\_specification  
;

PACKAGE\_name :  
name ;

package\_specification :  
PACKAGE Identifier IS ( declarative\_item )  
[ PRIVATE ( declarative\_item ) ]  
END [ Identifier ] ;

parameter\_declaration :  
Identifier\_list mode subtype\_indication [ COLON\_EQUALS ] expression ;

program\_component :  
body  
| package\_declaration  
;

subprogram\_body :  
subprogram\_specification IS declarative\_part  
BEGIN sequence\_of\_statements  
[ EXCEPTION ( exception\_handler ) ]  
END [ designator ] SEMICOLON ;

subprogram\_declaration\_body :  
subprogram\_specification  
(  
IS declarative\_part BEGIN sequence\_of\_statements  
[ EXCEPTION ( exception\_handler ) ]  
END [ designator ] SEMICOLON  
| SEMICOLON  
) ;

subprogram\_specification :  
PROCEDURE Identifier [ formal\_part ]  
| FUNCTION designator [ formal\_part ] RETURN subtype\_indication  
;

subtype\_indication : type\_mark [ constraint ] ;

type\_mark :  
TYPE\_name ;

TYPE\_name :  
name ;

UNIT\_name :  
name ;  
use\_clause :  
USE PACKAGE\_name ( COMMA PACKAGE\_name ) ;  
with\_clause :  
WITH UNIT\_name ( COMMA UNIT\_name ) ;

# Parser for Partial Ada Grammar

```

1  # define declaration
2  # define expression
3  # define sequence_of_statements
4  # define operator_symbol
5  # define constraint
6  # define identifier
7  # define discrete_range
8  # define parameter_association
9  # define exception_handler
10 # define designator
11 # define COLON_EQUALS
12 # define LEFT_PAREN
13 # define RIGHT_PAREN
14 # define COMMA
15 # define ENTRY
16 # define SEMICOLON
17 # define IN
18 # define OUT
19 # define DOT
20 # define ALL
21 # define BEGIN
22 # define PACKAGE
23 # define BODY
24 # define IS
25 # define EXCEPTION
26 # define END
27 # define PRIVATE
28 # define PROCEDURE
29 # define FUNCTION
30 # define RETURN
31 # define USE
32 # define WITH
33 # define END_OF_FILE
34 # define TRUE
35 # define FALSE
36
37 # ifndef LL_MAXCHAR
38 # define LL_MAXCHAR 255
39 # endif
40
41 char LL_text[LL_MAXCHAR+1]; /*contains the current token 'string'*/
42
43 int LL_length; /*contains the current token's length*/
44
45 /* standard output = listing and error messages
46    standard input = data to be read in */
47
48 /* LEXICAL ANALYZER */
49
50 static char LL_area[LL_MAXCHAR+1];
51
52 static int LL_lex()
53 {
54     static struct stack_type
55     {
56         int action;

```

```

char st_char;
}
stack [LL_MAXCHAR];
/*Each element will contain the action_id for a state,
and the character that came next. the transition to the
next state will be made on this character. The first
element will always have state 0's action id*/
int stackplace; /*points to the last used stack element*/
int transition_OK; /*TRUE while transitions can be made*/
int next_state; /*contains next state # after a transition*/
int i;
char c, LL_getc();
while (TRUE) /*will continue to execute until a return() is executed*/
{
    stackplace = 0;
    next_state = 0;
    transition_OK = TRUE;
    for (; transition_OK == TRUE; stackplace++)
    {
        stack[stackplace].st_char = c = LL_getc();
        if (c == EOF) stack[stackplace].st_char = c = END_OF_FILE;
        switch(next_state)
        {
            case 0 :
                stack[stackplace].action = 0;
                if ( ( c >= 'g' && c <= 'g' ) )
                    next_state = 4;
                else if ( c == '.' )
                    next_state = 1;
                else if ( ( c >= 'A' && c <= 'O' ) )
                    || ( c >= 'Q' && c <= 'Z' )
                    || ( c >= 'a' && c <= 'z' ) )
                        next_state = 2;
                else if ( c == 'p' )
                    next_state = 5;
                else transition_OK = FALSE;
                break;
            case 1 :
                stack[stackplace].action = 1;
                transition_OK = FALSE;
                break;
            case 2 :
                stack[stackplace].action = 2;
                if ( ( c >= 'A' && c <= 'Z' ) )
                    || ( c >= 'a' && c <= 'z' ) )
                        next_state = 3;
                else transition_OK = FALSE;
                break;
            case 3 :
                stack[stackplace].action = 2;
                if ( ( c >= 'A' && c <= 'Z' ) )
                    || ( c >= 'a' && c <= 'z' ) )
                        next_state = 3;
                else transition_OK = FALSE;
                break;
            case 4 :
                stack[stackplace].action = 3;
                if ( ( c >= '0' && c <= '9' ) )
                    next_state = 4;

```

```

else transition_OK = FALSE;
break;
case 5 :
stack[stackplace].action = 2;
if ( c == 'A' )
    next_state = 6;
else if ( ( c ) >= 'B' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else transition_OK = FALSE;
break;
case 6 :
stack[stackplace].action = 2;
if ( ( c ) >= 'A' && c <= 'B' )
    || ( c ) >= 'D' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else if ( c == 'C' )
    next_state = 7;
else transition_OK = FALSE;
break;
case 7 :
stack[stackplace].action = 2;
if ( ( c ) >= 'A' && c <= 'J' )
    || ( c ) >= 'L' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else if ( c == 'K' )
    next_state = 8;
else transition_OK = FALSE;
break;
case 8 :
stack[stackplace].action = 2;
if ( c == 'A' )
    next_state = 9;
else if ( ( c ) >= 'B' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else transition_OK = FALSE;
break;
case 9 :
stack[stackplace].action = 2;
if ( ( c ) >= 'A' && c <= 'F' )
    || ( c ) >= 'H' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else if ( c == 'G' )
    next_state = 10;
else transition_OK = FALSE;
break;
case 10 :
stack[stackplace].action = 2;
if ( ( c ) >= 'A' && c <= 'D' )
    || ( c ) >= 'F' && c <= 'Z' )
    || ( c ) >= 'a' && c <= 'z' )
    next_state = 3;
else if ( c == 'E' )
    next_state = 11;
else transition_OK = FALSE;

```

```

        break;
    case 11 :
        stack[stackplace].action = 2;
        if ( ( c >= 'A' && c <= 'Z' ) )
            if ( c >= 'a' && c <= 'z' ) )
                next_state = 3;
            else transition_OK = FALSE;
        break;
    };
}
/* the last state was the last good transition */
do
{
    stackplace--;
    LL_putc(stack[stackplace].st_char);
}
while(stackplace > 0 && stack[stackplace].action == 0);
if (stackplace == 0 && stack[stackplace].st_char == END_OF_FILE)
{
    LL_area[0] = '\0';
    return(END_OF_FILE);
};
if (stackplace == 0)
    LL_abort("character cannot be parsed");
for (i=0; i<stackplace; i++) LL_area[i] = stack[i].st_char;
LL_area[i] = '\0';
switch(stack[stackplace].action)
{
    case 1 :
        return(COLON);
        break;
    case 2 :
        return(identifier);
        break;
    case 3 :
        return(expression);
        break;
    case 4 :
        return(PACKAGE);
        break;
};
}

/* SUBROUTINES FOR LEXICAL ANALYZER */

static char LL_input_line[LL_MAXCHAR*2];
static int LL_input_length = 0;
static int LL_curchar = 1;
static int LL_errorchar;

static char LL_getc()
{
    if (LL_curchar >= LL_input_length)
        getline();
    LL_curchar = 0;
    LL_errorchar = 0;

```

```

    );
    LL_errorchar++;
    LL_curchar++;
    return(LL_input_line[LL_curchar-1]);
}

static LL_putc(c)
char c;
{
    int i;
    if (LL_curchar == 0)
    {
        if (LL_input_length == LL_MAXCHAR*2)
            LL_abort("put too many chars back. Compiler abort");
        for (i = LL_input_length; i > 0; i--)
            LL_input_line[i] = LL_input_line[i-1];
        LL_input_length++;
        LL_curchar = 1;
    }
    LL_errorchar--;
    LL_curchar--;
    LL_input_line[LL_curchar] = c;
}

static getline()
{
    for (LL_input_length = 0;
         LL_input_length < LL_MAXCHAR*2 &&
         (LL_input_line[LL_input_length] = getchar()) != EOF;
         LL_input_length++)
    {
        if (LL_input_line[LL_input_length] == '\n')
            break;
    };
    if (LL_input_length == LL_MAXCHAR*2)
    {
        LL_errorchar = -1;
        LL_abort("next LL_input_line too long");
    };
    LL_input_length++;
    LL_input_line[LL_input_length] = '\0';
    if (LL_input_line[LL_input_length-1] == EOF)
    {
        int i;
        for (i=0; i<LL_input_length-1; i++)
            printf("%c", LL_input_line[i]);
        printf("\n");
    }
    else
    {
        printf(LL_input_line);
    }
}

```

```

# define LL_k 2
compile()
{

```

```

compilation(1);
if(LL_lookahead(1) != END_OF_FILE)
    LL_syntax_error(LL_text);
}

compilation(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    while (TRUE)
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == WITH )
            || ( LL_lookahead(1) == PACKAGE )
            || ( LL_lookahead(1) == PROCEDURE )
            || ( LL_lookahead(1) == FUNCTION ) )
        {
            set_begin = LL_begin_set();
            compilation_unit(1);
            LL_end_set(set_begin,TRUE,first_closure,FALSE);
        }
        else
        {
            break;
        }
    };
    LL_end_set(set_begin,FALSE,first_closure,TRUE);
}

compilation_unit(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    context_specification(1);
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        {
            int *set_begin,*first_closure;
            first_closure = NULL;
            set_begin = LL_begin_set();
            if ( ( LL_lookahead(1) == FUNCTION )
                || ( LL_lookahead(1) == PROCEDURE ) )
            {
                subprogram_declaration_body(1);
                LL_end_set(set_begin,FALSE,first_closure,FALSE);
            }
            else

```



```

{
  if ( ( LL_lookahead(1) == PACKAGE &&
        LL_lookahead(2) == Identifier))
  {
    package_declaration(1);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
  }
  else {
    package_body(1);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
  };
  LL_end_set(set_begin,FALSE,first_closure,FALSE);
};
LL_end_set(set_begin,FALSE,first_closure,FALSE);
}

body(LL_reference)
int LL_reference;
{
  int *set_begin,*first_closure;
  int *LL_begin_set(),*LL_begin_closure();
  first_closure = NULL;
  set_begin = LL_begin_set();
  if ( ( LL_lookahead(1) == FUNCTION)
        || ( LL_lookahead(1) == PROCEDURE) )
  {
    subprogram_body(1);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
  }
  else {
    package_body(2);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
  }
}

subprogram_body(LL_reference)
int LL_reference;
{
  int *set_begin,*first_closure;
  int *LL_begin_set(),*LL_begin_closure();
  first_closure = NULL;
  set_begin = LL_begin_set();
  subprogram_specification(1);
  if (IS != LL_get_token())
    LL_syntax_error(LL_text);
  LL_new_entry(IS);
  declarative_part(1);
  if (BEGIN != LL_get_token())
    LL_syntax_error(LL_text);
  LL_new_entry(BEGIN);
  if (sequence_of_statements != LL_get_token())
    LL_syntax_error(LL_text);
  LL_new_entry(sequence_of_statements);
  if ( first_closure == NULL )
    first_closure = LL_begin_closure();
}

```

```

else LL_begin_closure();
if ( ( LL_lookahead() == EXCEPTION))
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (EXCEPTION != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(EXCEPTION);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    while (TRUE)
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead() == exception_handler))
        {
            set_begin = LL_begin_set();
            if (exception_handler != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(exception_handler);
            LL_end_set(set_begin,TRUE,first_closure,FALSE);
        }
        else
        {
            break;
        }
    }
    LL_end_set(set_begin,TRUE,first_closure,TRUE);
};
if (END != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(END);
if ( first_closure == NULL )
    first_closure = LL_begin_closure();
else LL_begin_closure();
if ( ( LL_lookahead() == designator))
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (designator != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(designator);
    LL_end_set(set_begin,TRUE,first_closure,FALSE);
};
if (SEMICOLON != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(SEMICOLON);
LL_end_set(set_begin,FALSE,first_closure,FALSE);
}

package_body(LL_reference)
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;

```

```

set_begin = LL_begin_set();
if (PACKAGE != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(PACKAGE);
if (BODY != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(BODY);
if (Identifier != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(Identifier);
if (IS != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(IS);
declarative_part(2);
if ( first_closure == NULL )
    first_closure = LL_begin_closure();
else LL_begin_closure();
if ( ( LL_lookahead(1) == BEGIN )
    (
        int *set_begin,*first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        if (BEGIN != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(BEGIN);
        if (sequence_of_statements != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(sequence_of_statements);
        if ( first_closure == NULL )
            first_closure = LL_begin_closure();
        else LL_begin_closure();
        if ( ( LL_lookahead(1) == EXCEPTION )
            (
                int *set_begin,*first_closure;
                first_closure = NULL;
                set_begin = LL_begin_set();
                if (EXCEPTION != LL_get_token())
                    LL_syntax_error(LL_text);
                LL_new_entry(EXCEPTION);
                if ( first_closure == NULL )
                    first_closure = LL_begin_closure();
                else LL_begin_closure();
                while (TRUE)
                {
                    int *set_begin,*first_closure;
                    first_closure = NULL;
                    if ( ( LL_lookahead(1) == exception_handler )
                        (
                            set_begin = LL_begin_set();
                            if (exception_handler != LL_get_token())
                                LL_syntax_error(LL_text);
                            LL_new_entry(exception_handler);
                            LL_end_set(set_begin,TRUE,first_closure,FALSE);
                        )
                    )
                    else
                    {
                        break;
                    }
                }
            )
        )
    )

```

```

        LL_end_set(set_begin, TRUE, first_closure, TRUE);
    };
    LL_end_set(set_begin, TRUE, first_closure, TRUE);
};
if (END != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(END);
if (first_closure == NULL)
    first_closure = LL_begin_closure();
else LL_begin_closure();
if ( (LL_lookahead() == identifier)
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        if (identifier != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(identifier);
        LL_end_set(set_begin, TRUE, first_closure, FALSE);
    };
if (SEMICOLON != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(SEMICOLON);
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}

context_specification(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (first_closure == NULL)
        first_closure = LL_begin_closure();
    while (TRUE)
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        if ( (LL_lookahead() == WITH))
        {
            set_begin = LL_begin_set();
            with_clause();
            if (first_closure == NULL)
                first_closure = LL_begin_closure();
            else LL_begin_closure();
            while (TRUE)
            {
                int *set_begin, *first_closure;
                first_closure = NULL;
                if ( (LL_lookahead() == USE))
                {
                    set_begin = LL_begin_set();
                    use_clause();
                    LL_end_set(set_begin, TRUE, first_closure, FALSE);
                }
                else
            {

```

```

        break;
    };
    LL_end_set(set_begin, TRUE, first_closure, TRUE);
}
else {
    break;
};
LL_end_set(set_begin, FALSE, first_closure, TRUE);
}

subprogram_declaration_body(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    subprogram_specification(2);
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        {
            int *set_begin, *first_closure;
            first_closure = NULL;
            set_begin = LL_begin_set();
            if ( ( LL_lookahead(1) == IS ) )
            {
                if ( IS != LL_get_token() )
                    LL_syntax_error(LL_text);
                LL_new_entry(IS);
                declarative_part(3);
                if ( BEGIN != LL_get_token() )
                    LL_syntax_error(LL_text);
                LL_new_entry(BEGIN);
                if (sequence_of_statements != LL_get_token() )
                    LL_syntax_error(LL_text);
                LL_new_entry(sequence_of_statements);
                if ( first_closure == NULL )
                    first_closure = LL_begin_closure();
                else LL_begin_closure();
                if ( ( LL_lookahead(1) == EXCEPTION ) )
                {
                    int *set_begin, *first_closure;
                    first_closure = NULL;
                    set_begin = LL_begin_set();
                    if ( EXCEPTION != LL_get_token() )
                        LL_syntax_error(LL_text);
                    LL_new_entry(EXCEPTION);
                    if ( first_closure == NULL )
                        first_closure = LL_begin_closure();
                    else LL_begin_closure();
                    while (TRUE)
                    {
                        int *set_begin, *first_closure;
                        first_closure = NULL;

```

```

if ( ( LL_lookahead(1) == exception_handler))
{
    set_begin = LL_begin_set();
    if (exception_handler != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(exception_handler);
    LL_end_set(set_begin, TRUE, first_closure, FALSE);
}
else
{
    break;
};
LL_end_set(set_begin, TRUE, first_closure, TRUE);
};
if (END != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(END);
if ( first_closure == NULL )
    first_closure = LL_begin_closure();
else LL_begin_closure();
if ( ( LL_lookahead(1) == designator))
{
    int *set_begin, *first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (designator != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(designator);
    LL_end_set(set_begin, TRUE, first_closure, FALSE);
};
if (SEMICOLON != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(SEMICOLON);
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}
else
{
    if (SEMICOLON != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(SEMICOLON);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
};
LL_end_set(set_begin, FALSE, first_closure, FALSE);
};
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}
package_declaration(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    package_specification(1);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
}

```

```

with_clause(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (WITH != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(WITH);
    UNIT_name(1);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    else LL_begin_closure();
    while (TRUE)
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == COMMA ))
        {
            set_begin = LL_begin_set();
            if (COMMA != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(COMMA);
            UNIT_name(2);
            LL_end_set(set_begin,TRUE,first_closure,FALSE);
        }
        else
        {
            break;
        }
    };
    LL_end_set(set_begin,FALSE,first_closure,TRUE);
}

use_clause(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (USE != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(USE);
    PACKAGE_name(1);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    else LL_begin_closure();
    while (TRUE)
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == COMMA ))
        {
            set_begin = LL_begin_set();
            if (COMMA != LL_get_token())
                LL_syntax_error(LL_text);
        }
    }
}

```

```

LL_new_entry(COMMA);
PACKAGE_name(2);
LL_end_set(set_begin, TRUE, first_closure, FALSE);
}
else
{
break;
};
LL_end_set(set_begin, FALSE, first_closure, TRUE);
}

declarative_item(LL_reference)
int LL_reference;
{
int *set_begin, *first_closure;
int *LL_begin_set(), *LL_begin_closure();
first_closure = NULL;
set_begin = LL_begin_set();
if ( ( LL_lookahead(1) == declaration )
{
if ( declaration != LL_get_token() )
LL_syntax_error(LL_text);
LL_new_entry(declaration);
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}
else
{
use_clause(2);
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}
}

declarative_part(LL_reference)
int LL_reference;
{
int *set_begin, *first_closure;
int *LL_begin_set(), *LL_begin_closure();
first_closure = NULL;
set_begin = LL_begin_set();
if ( first_closure == NULL )
first_closure = LL_begin_closure();
else LL_begin_closure();
while (TRUE)
{
int *set_begin, *first_closure;
first_closure = NULL;
if ( ( LL_lookahead(1) == USE )
|| ( LL_lookahead(1) == declaration )
{
set_begin = LL_begin_set();
declarative_item(1);
LL_end_set(set_begin, TRUE, first_closure, FALSE);
}
else
{
break;
};
}
}

```



```

if ( first_closure == NULL )
    first_closure = LL_begin_closure();
else LL_begin_closure();
while (TRUE)
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    if ( ( LL_lookahead(1) == FUNCTION )
        || ( LL_lookahead(1) == PROCEDURE )
        || ( LL_lookahead(1) == PACKAGE ) )
    {
        set_begin = LL_begin_set();
        program_component(1);
        LL_end_set(set_begin,TRUE,first_closure,FALSE);
    }
    else
    {
        break;
    }
};
LL_end_set(set_begin,FALSE,first_closure,TRUE);
}

program_component(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if ( ( LL_lookahead(1) == FUNCTION &&
        LL_lookahead(2) == designator )
        || ( LL_lookahead(1) == PROCEDURE &&
        LL_lookahead(2) == identifier )
        || ( LL_lookahead(1) == PACKAGE &&
        LL_lookahead(2) == BODY ) )
    {
        body(1);
        LL_end_set(set_begin,FALSE,first_closure,FALSE);
    }
    else
    {
        package_declaration(2);
        LL_end_set(set_begin,FALSE,first_closure,FALSE);
    }
}

formal_part(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (LEFT_PAREN != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(LEFT_PAREN);
    parameter_declaration(1);
    if ( first_closure == NULL )

```

```

        fir_t_closure = LL_begin_closure();
    else LL_begin_closure();
    while (TRUE)
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == SEMICOLON) )
        {
            set_begin = LL_begin_set();
            if (SEMICOLON != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(SEMICOLON);
            parameter_declaration(2);
            LL_end_set(set_begin, TRUE, first_closure, FALSE);
        }
        else
        {
            break;
        }
    };
    if (RIGHT_PAREN != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(RIGHT_PAREN);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
}

parameter_declaration(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    identifier_list(1);
    mode(1);
    subtype_indication(1);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    else LL_begin_closure();
    if ( ( LL_lookahead(1) == COLON_EQUALS) )
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        if (COLON_EQUALS != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(COLON_EQUALS);
        LL_end_set(set_begin, TRUE, first_closure, FALSE);
    };
    if (expression != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(expression);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
}

identifier_list(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;

```

```

int *LL_begin_set(), *LL_begin_closure();
first_closure = NULL;
set_begin = LL_begin_set();
if (identifier != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(identifier);
if (first_closure == NULL)
    else first_closure = LL_begin_closure();
while (TRUE)
{
    int *set_begin, *first_closure;
    first_closure = NULL;
    if ( ( LL_lookahead(1) == COMMA ) )
    {
        set_begin = LL_begin_set();
        if (COMMA != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(COMMA);
        if (identifier != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(identifier);
        LL_end_set(set_begin, TRUE, first_closure, FALSE);
    }
    else
    {
        break;
    }
};
LL_end_set(set_begin, FALSE, first_closure, TRUE);

mode(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if ( ( LL_reference == 1 &&
        LL_lookahead(1) == identifier &&
        LL_lookahead(2) == LEFT_PAREN )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == identifier &&
        LL_lookahead(2) == DOT )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == identifier &&
        LL_lookahead(2) == constraint )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == identifier &&
        LL_lookahead(2) == COLON_EQUALS )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == identifier &&
        LL_lookahead(2) == expression )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == operator_symbol &&
        LL_lookahead(2) == LEFT_PAREN )
        || ( LL_reference == 1 &&
        LL_lookahead(1) == operator_symbol &&

```

```

        LL_lookahead(2) == DOT)
    || ( LL_reference == 1 &&
        LL_lookahead(1) == operator_symbol &&
        LL_lookahead(2) == constraint)
    || ( LL_reference == 1 &&
        LL_lookahead(1) == operator_symbol &&
        LL_lookahead(2) == COLON_EQUALS)
    || ( LL_reference == 1 &&
        LL_lookahead(1) == operator_symbol &&
        LL_lookahead(2) == expression)
    || ( LL_lookahead(1) == IN &&
        LL_reference == 1 &&
        LL_lookahead(2) == identifier)
    || ( LL_lookahead(1) == IN &&
        LL_reference == 1 &&
        LL_lookahead(2) == operator_symbol))
    {
        if ( first_closure == NULL )
            first_closure = LL_begin_closure();
        else LL_begin_closure();
        if ( ( LL_lookahead(1) == IN))
        {
            int *set_begin,*first_closure;
            first_closure = NULL;
            set_begin = LL_begin_set();
            if (IN != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(IN);
            LL_end_set(set_begin,TRUE,first_closure,FALSE);
        }
        LL_end_set(set_begin,FALSE,first_closure,TRUE);
    }
    else
    {
        if ( ( LL_lookahead(1) == OUT))
        {
            if (OUT != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(OUT);
            LL_end_set(set_begin,FALSE,first_closure,FALSE);
        }
        else
        {
            if (IN != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(IN);
            if (OUT != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(OUT);
            LL_end_set(set_begin,FALSE,first_closure,FALSE);
        }
    }
}

name(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();

```

```

first_closure = NULL;
set_begin = LL_begin_set();

int *set_begin,*first_closure;
first_closure = NULL;
set_begin = LL_begin_set();
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    if ( { LL_lookahead(1) == identifier })
    {
        if ( identifier != LL_get_token() )
            LL_syntax_error(LL_text);
        LL_new_entry(identifier);
        LL_end_set(set_begin,FALSE,first_closure,FALSE);
    }
    else
    {
        if ( operator_symbol != LL_get_token() )
            LL_syntax_error(LL_text);
        LL_new_entry(operator_symbol);
        LL_end_set(set_begin,FALSE,first_closure,FALSE);
    }
};
LL_end_set(set_begin,FALSE,first_closure,FALSE);
};
name_l(1);
LL_end_set(set_begin,FALSE,first_closure,FALSE);

name_l(LL_reference)
int LL_reference;

int *set_begin,*first_closure;
int *LL_begin_set(),*LL_begin_closure();
first_closure = NULL;
set_begin = LL_begin_set();
if ( { LL_lookahead(1) == LEFT_PAREN }
    || { LL_lookahead(1) == DOT } )
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        if ( { LL_lookahead(1) == LEFT_PAREN &&
            LL_lookahead(2) == expression } )
        {
            if ( LEFT_PAREN != LL_get_token() )
                LL_syntax_error(LL_text);
            LL_new_entry(LEFT_PAREN);
            if ( expression != LL_get_token() )
                LL_syntax_error(LL_text);
            LL_new_entry(expression);
            if ( first_closure == NULL )

```

```

first_closure = LL_begin_closure();
else LL_begin_closure();
while (TRUE)
{
    int *set_begin, *first_closure;
    first_closure = NULL;
    if ( ( LL_lookahead(1) == COMMA ) )
    {
        set_begin = LL_begin_set();
        if (COMMA != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(COMMA);
        if (expression != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(expression);
        LL_end_set(set_begin, TRUE, first_closure, FALSE);
    }
    else
    {
        break;
    }
};
if (RIGHT_PAREN != LL_get_token())
    LL_syntax_error(LL_text);
LL_new_entry(RIGHT_PAREN);
LL_end_set(set_begin, FALSE, first_closure, FALSE);
}
else
{
    if ( ( LL_lookahead(1) == LEFT_PAREN &&
          LL_lookahead(2) == discrete_range ) )
    {
        if (LEFT_PAREN != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(LEFT_PAREN);
        if (discrete_range != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(discrete_range);
        if (RIGHT_PAREN != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(RIGHT_PAREN);
        LL_end_set(set_begin, FALSE, first_closure, FALSE);
    }
    else
    {
        if ( ( LL_lookahead(1) == DOT &&
              LL_lookahead(2) == identifier ) )
        {
            if (DOT != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(DOT);
            if (identifier != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(identifier);
            LL_end_set(set_begin, FALSE, first_closure, FALSE);
        }
        else
        {
            if ( ( LL_lookahead(1) == DOT &&
                  LL_lookahead(2) == DOT ) )
            {
                if (DOT != LL_get_token())
                    LL_syntax_error(LL_text);
                LL_new_entry(DOT);
                if (DOT != LL_get_token())
                    LL_syntax_error(LL_text);
                LL_new_entry(DOT);
                LL_end_set(set_begin, FALSE, first_closure, FALSE);
            }
        }
    }
}

```

```

        LL_lookahead(2) == ALL))
    {
        if (DOT != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(DOT);
        if (ALL != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(ALL);
        LL_end_set(set_begin, FALSE, first_closure, FALSE);
    }
    else
    {
        if ( ( LL_lookahead(1) == DOT))
        {
            if (DOT != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(DOT);
            if (operator_symbol != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(operator_symbol);
            LL_end_set(set_begin, FALSE, first_closure, FALSE);
        }
        else
        {
            if (LEFT_PAREN != LL_get_token())
                LL_syntax_error(LL_text);
            LL_new_entry(LEFT_PAREN);
            if ( first_closure == NULL )
                first_closure = LL_begin_closure();
            else LL_begin_closure();
            if ( ( LL_lookahead(1) == parameter_association))
            {
                int *set_begin, *first_closure;
                first_closure = NULL;
                set_begin = LL_begin_set();
                if (parameter_association != LL_get_token())
                    LL_syntax_error(LL_text);
                LL_new_entry(parameter_association);
                if ( first_closure == NULL )
                    first_closure = LL_begin_closure();
                else LL_begin_closure();
                while (TRUE)
                {
                    int *set_begin, *first_closure;
                    first_closure = NULL;
                    if ( ( LL_lookahead(1) == COMMA))
                    {
                        set_begin = LL_begin_set();
                        if (COMMA != LL_get_token())
                            LL_syntax_error(LL_text);
                        LL_new_entry(COMMA);
                        if (parameter_association != LL_get_token())
                            LL_syntax_error(LL_text);
                        LL_new_entry(parameter_association);
                        LL_end_set(set_begin, TRUE, first_closure, FALSE);
                    }
                    else
                    {
                        break;
                    }
                }
            }
        }
    }
}

```

```

    );
    LL_end_set(set_begin, TRUE, first_closure, TRUE);
    );
    if (RIGHT_PAREN != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(RIGHT_PAREN);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
    );
    );
    );
    );
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
    name_1(2);
    LL_end_set(set_begin, FALSE, first_closure, FALSE);
    }
    else {
        LL_end_set(set_begin, FALSE, first_closure, FALSE);
    }
}

package specification(LL_reference)
int LL_reference;
{
    int *set_begin, *first_closure;
    int *LL_begin_set(), *LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (PACKAGE != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(PACKAGE);
    if (Identifier != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(Identifier);
    if (IS != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(IS);
    if (first_closure == NULL)
        first_closure = LL_begin_closure();
    else LL_begin_closure = LL_begin_closure();
    while (TRUE)
    {
        int *set_begin, *first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == USE )
            || ( LL_lookahead(1) == declaration ) )
        {
            set_begin = LL_begin_set();
            declarative_item(2);
            LL_end_set(set_begin, TRUE, first_closure, FALSE);
        }
        else {
            break;
        }
    }
}

```



```

);
if ( first_closure == NULL )
    first_closure = LL_begin_closure();
else LL_begin_closure();
if ( ( LL_lookahead(1) == PRIVATE ) )
{
    int *set_begin,*first_closure;
    first_closure = NULL;
    set_begin = LL_begin_set();
    if (PRIVATE != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(PRIVATE);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    else LL_begin_closure();
    while (TRUE)
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        if ( ( LL_lookahead(1) == USE ) )
        {
            if ( ( LL_lookahead(1) == declaration ) )
            {
                set_begin = LL_begin_set();
                declarative_item(3);
                LL_end_set(set_begin,TRUE,first_closure,FALSE);
            }
            else
            {
                break;
            }
        }
        LL_end_set(set_begin,TRUE,first_closure,TRUE);
    };
    if (END != LL_get_token())
        LL_syntax_error(LL_text);
    LL_new_entry(END);
    if ( first_closure == NULL )
        first_closure = LL_begin_closure();
    else LL_begin_closure();
    if ( ( LL_lookahead(1) == identifier ) )
    {
        int *set_begin,*first_closure;
        first_closure = NULL;
        set_begin = LL_begin_set();
        if (identifier != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(identifier);
        LL_end_set(set_begin,TRUE,first_closure,FALSE);
    };
    LL_end_set(set_begin,FALSE,first_closure,TRUE);
}

PACKAGE_name(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();

```

```

name(1);
LL_end_set(set_begin,FALSE,first_closure,FALSE);
)

subtype_indication(LL_reference)
int LL_reference;
(
  int *set_begin,*first_closure;
  int *LL_begin_set(),*LL_begin_closure();
  first_closure = NULL;
  set_begin = LL_begin_set();
  type_mark(1);
  if ( first_closure == NULL )
    first_closure = LL_begin_closure();
  else LL_begin_closure();
  if ( ( LL_lookahead(1) == constraint )
    (
      int *set_begin,*first_closure;
      first_closure = NULL;
      set_begin = LL_begin_set();
      if (constraint != LL_get_token())
        LL_syntax_error(LL_text);
      LL_new_entry(constraint);
      LL_end_set(set_begin,TRUE,first_closure,FALSE);
    );
    LL_end_set(set_begin,FALSE,first_closure,TRUE);
  )

subprogram_specification(LL_reference)
int LL_reference;
(
  int *set_begin,*first_closure;
  int *LL_begin_set(),*LL_begin_closure();
  first_closure = NULL;
  set_begin = LL_begin_set();
  if ( ( LL_lookahead(1) == PROCEDURE )
    (
      if (PROCEDURE != LL_get_token())
        LL_syntax_error(LL_text);
      LL_new_entry(PROCEDURE);
      if (identifier != LL_get_token())
        LL_syntax_error(LL_text);
      LL_new_entry(identifier);
      if ( first_closure == NULL )
        first_closure = LL_begin_closure();
      else LL_begin_closure();
      if ( ( LL_lookahead(1) == LEFT_PAREN )
        (
          int *set_begin,*first_closure;
          first_closure = NULL;
          set_begin = LL_begin_set();
          formal_part(1);
          LL_end_set(set_begin,TRUE,first_closure,FALSE);
        );
        LL_end_set(set_begin,FALSE,first_closure,TRUE);
      )
    )
    else
    (
      if (FUNCTION != LL_get_token())

```

```

        LL_syntax_error(LL_text);
        LL_new_entry(FUNCTION);
        if (designator != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(designator);
        if ( first_closure == NULL )
            first_closure = LL_begin_closure();
        else LL_begin_closure();
        if ( { LL_lookahead(1) == LEFT_PAREN } )
        {
            int *set_begin,*first_closure;
            first_closure = NULL;
            set_begin = LL_begin_set();
            formal_part(2);
            LL_end_set(set_begin,TRUE,first_closure,FALSE);
        };
        if (RETURN != LL_get_token())
            LL_syntax_error(LL_text);
        LL_new_entry(RETURN);
        subtype_indication(2);
        LL_end_set(set_begin,FALSE,first_closure,FALSE);
    };
}

type_mark(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    TYPE_name(1);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
}

TYPE_name(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    name(2);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
}

UNIT_name(LL_reference)
int LL_reference;
{
    int *set_begin,*first_closure;
    int *LL_begin_set(),*LL_begin_closure();
    first_closure = NULL;
    set_begin = LL_begin_set();
    name(3);
    LL_end_set(set_begin,FALSE,first_closure,FALSE);
}

```

```

/* SUBROUTINES FOR PARSER */

static struct LL_table
/* #th entry not used */
int token_num;
char token_name[LL_MAXCHAR];
}
LL_lookahead_table[LL_k+1];

static int LL_num_lookahead = 0;

static int LL_get_token()
{
    int i, token;
    if (LL_num_lookahead)
    {
        token = LL_lookahead_table[i].token_num;
        LL_strcpy(LL_lookahead_table[i].token_name, LL_text);
        for (i=1; i<LL_num_lookahead; i++)
        {
            LL_lookahead_table[i].token_num =
            LL_lookahead_table[i+1].token_num;
            LL_strcpy(LL_lookahead_table[i+1].token_name,
            LL_lookahead_table[i].token_name);
        };
        LL_num_lookahead--;
    }
    else
    {
        token = LL_lex();
        LL_strcpy(LL_area, LL_text);
    };
    LL_length = strlen(LL_text);
    return(token);
}

static int LL_lookahead(n)
int n;
{
    int token;
    if (n > LL_num_lookahead)
    {
        if (n > 1 && LL_lookahead_table[n-1].token_num == END_OF_FILE)
        {
            LL_lookahead_table[n].token_num = END_OF_FILE;
        }
        else
        {
            LL_lookahead_table[n].token_num = token = LL_lex();
            LL_strcpy(LL_area, LL_lookahead_table[n].token_name);
        };
        LL_num_lookahead++;
    }
    else
    {
        token = LL_lookahead_table[n].token_num;
    };
    return(token);
}

```

```

)

/* SUBROUTINES FOR VALUE STACK IN PARSER */

# ifndef LL_VAL_STACK
# define LL_VAL_STACK 10000
# endif

static int LL_val [LL_VAL_STACK];
static int LL_cval[LL_VAL_STACK];
static int *LL_top = &(LL_val [0]);
static int *LL_ctop = &(LL_cval[0]);
static int *LL_bound = &(LL_val [LL_VAL_STACK-1]);
static int *LL_cbound = &(LL_cval[LL_VAL_STACK-1]);

static int *
LL_begin_closure()
{
    *LL_top++ = (int) LL_ctop;
    *LL_ctop++ = 0;
    LL_check_stacks();
    return(LL_ctop-1);
}

static int *
LL_begin_set()
{
    LL_top++;
    LL_check_stacks();
    return(LL_top-1);
}

static
LL_new_entry(value)
int value;
{
    *LL_top++ = value;
    LL_check_stacks();
}

static
LL_end_set(set_begin,in_closure,first_closure,last_item_closure)
int *set_begin;
int in_closure;
int *first_closure;
int last_item_closure;
{
    int set_value;
    if (last_item_closure)
    {
        if (*(LL_top-1) == 0)
            set_value = 0;
        else
            set_value = *(LL_ctop-1);
    }
    else
        set_value = *(LL_top-1);
    if (first_closure != NULL)
        LL_ctop = first_closure;
}

```

```

if (in_closure)
{
    *LL_ctype++ = set_value;
    LL_top = set_begin;
    *(((int *)*(LL_top-1)) = *(((int *)*(LL_top-1))))+1;
}

else
{
    LL_top = set_begin;
    *LL_ctype++ = set_value;
};
LL_check_stacks();

static LL_check_stacks()
{
    if (LL_top > LL_bound || LL_ctype > LL_cbound )
        LL_abort("LL error. Value stack overflow.");
}

/* GENERAL SUBROUTINES */

LL_abort(str)
char *str;
{
    int i;
    ifdef LL_error_char
    for (i=0; i < LL_errorchar; i++)
        printf("%c", LL_error_char[i]);
    printf("\n%s\n", str);
    endif
    exit(0);
}

static LL_syntax_error(token)
char *token;
{
    char message[LL_MAXCHAR+31];
    sprintf(message, "syntax error. The token was %s.", token);
    LL_abort(message);
}

static
LL_strcpy(t,s)
char *s,*t;
{
    while (*s++ = *t++);
}

LL_print_stacks(set_begin,first_closure)
int *set_begin,*first_closure;
{
    extern int *LL_top,*LL_ctype,LL_val[],LL_cval[];
    int *i,*j,k;
    printf("\n");
    for (i = &(LL_val[0]), j = &(LL_cval[0]), k=0;
        i < LL_top || j < LL_ctype; i++, j++, k++)
    {
        if (i < LL_top)

```

```

(
    printf("X3d X5d %.k.*i");
    if (set_begin == i)
        printf("set_begin");
    else
        printf(" ");
    )
    else
        printf(" ");
        printf(" | ");
        if (j < LL_ctop)
        {
            printf("X3d X5d %.k.*j");
            if (first_closure == j)
                printf("first_closure");
            else
                printf(" ");
            )
            else
                printf(" ");
                printf("\n");
                printf("\n");
                fflush(stdout);
        }
    );

```

# LL Output File

TOKEN LIST

declaration	1
expression	2
sequence_of_statements	3
operator_symbol	4
constant	5
identifier	6
discrete_range	7
parameter_declaration	8
exception_handler	9
designator	10
COLON	11
COLON_EQUALS	12
LEFT_PAREN	13
RIGHT_PAREN	14
COMMA	15
END	16
Semicolon	17
IF	18
THEN	19
DO	20
ALL	21
PERIOD	22
PACKAGE	23
Body	24
IS	25
EXCEPTION	26
END	27
PRIVATE	28
PROCEDURE	29
FUNCTION	30
RETURN	31
USE	32
WITH	33
END FOREN LIST	

## STATE LIST

0	1	0
1	1	4
2	1	4
3	1	4
4	1	4
5	1	4
6	1	4
7	1	4
8	1	4
9	1	4
10	1	4
11	1	4
12	1	4
13	1	4
14	1	4
15	1	4
16	1	4
17	1	4
18	1	4
19	1	4
20	1	4
21	1	4
22	1	4
23	1	4
24	1	4
25	1	4
26	1	4
27	1	4
28	1	4
29	1	4
30	1	4
31	1	4
32	1	4
33	1	4





3  
= = = = =  
.L'.M'.N'.O'.P'.Q'.R'.S'.T'.U'.V'.W'.X'.Y'.Z'.a'.b'.c'.d'.e'.f'.g'.h'.i'.j'.k'.l'.m'.n'.o'.p'.q'.r'.s'.t'.u'.v'.w'.x'.y'.z'.  
2  
A'.B'.C'.D'.E'.F'.G'.H'.I'.J'.K'.L'.M'.N'.O'.P'.Q'.





[illegible]

[illegible]









```
'w' = 3
'x' = 3
'y' = 3
'z' = 3
```

# END STATE LIST

```
compilation_unit is not LL(1)
program_component is not LL(1)
mode is not LL(1)
name_1 is not LL(1)
name_1 is not LL(1)
name_1 is not LL(1)
name_1 is not LL(1)
```

## PRODUCTION TREES

\*\*\*\*\*

```
compilation
! * k = 1
! First Set
! Follow Set
! ! compilation_unit k = 0 ref = 1
! ! First Set
! ! - WITH(0)
! ! - PACKAGE(0)
! ! - PROCEDURE(0)
! ! - FUNCTION(0)
! ! Follow Set
```

\*\*\*\*\*

```
compilation_unit
! o k = 0
! First Set
! Follow Set
! ! context_specification k = 0 ref = 1
! ! First Set
! ! Follow Set
! ! o k = 0
! ! First Set
! ! Follow Set
! ! o k = 0
! ! First Set
! ! Follow Set
! ! empty k = 0
! ! First Set
! ! Follow Set
! ! k = 1
! ! First Set
! ! Follow Set
! ! subprogram_declaration_body k = 0 ref = 1
! ! First Set
! ! - FUNCTION(0)
! ! - PROCEDURE(0)
! ! Follow Set
! ! k = 2
! ! First Set
! ! - PACKAGE(0)
```

```

!! !! !! Follow Set
!! !! !! ! package_declaration k = Ø ref = 1
!! !! !! First Set
!! !! !! - PACKAGE(Ø) Identifier(Ø)
!! !! !! Follow Set
!! !! !! ! package_body k = Ø ref = 1
!! !! !! First Set
!! !! !! - PACKAGE(Ø) BODY(Ø)
!! !! !! Follow Set
!! !! !! ! empty k = Ø
!! !! !! First Set
!! !! !! Follow Set

```

\*\*\*\*\*

```

body
!! !! k = 1
!! !! First Set
!! !! Follow Set
!! !! ! subprogram_body k = Ø ref = 1
!! !! First Set
!! !! - FUNCTION(Ø)
!! !! - PROCEDURE(Ø)
!! !! Follow Set
!! !! ! package_body k = Ø ref = 2
!! !! First Set
!! !! - PACKAGE(Ø)
!! !! Follow Set

```

\*\*\*\*\*

```

subprogram_body
!! !! k = Ø
!! !! First Set
!! !! Follow Set
!! !! ! subprogram_specification k = Ø ref = 1
!! !! First Set
!! !! Follow Set
!! !! o k = Ø
!! !! First Set
!! !! Follow Set
!! !! ! IS k = Ø
!! !! First Set
!! !! Follow Set
!! !! o k = Ø
!! !! First Set
!! !! Follow Set
!! !! ! declarative_part k = Ø ref = 1
!! !! First Set
!! !! Follow Set
!! !! o k = Ø
!! !! First Set
!! !! Follow Set
!! !! ! BEGIN k = Ø
!! !! First Set
!! !! Follow Set
!! !! o k = Ø
!! !! First Set
!! !! Follow Set

```

```

sequence_of_statements k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
? k = 1
First Set
Follow Set
o k = Ø
First Set
- EXCEPTION(Ø)
Follow Set
EXCEPTION k = Ø
First Set
Follow Set
* k = 1
First Set
Follow Set
exception_handler k = Ø
First Set
- exception_handler(Ø)
Follow Set
o k = Ø
First Set
Follow Set
END k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
? k = 1
First Set
Follow Set
designator k = Ø
First Set
- designator(Ø)
Follow Set
SEMICOLON k = Ø
First Set
Follow Set

```

\*\*\*\*\*

```

package_body
! o k = Ø
! First Set
! Follow Set
! ! PACKAGE k = Ø
! First Set
! Follow Set
! o k = Ø
! First Set
! Follow Set
! ! BODY k = Ø
! First Set
! Follow Set
! ! o k = Ø

```

```

First Set
Follow Set
Identifier k = 0
First Set
Follow Set
o k = 0
First Set
Follow Set
IS k = 0
First Set
Follow Set
o k = 0
First Set
Follow Set
declarative_part k = 0 ref = 2
First Set
Follow Set
o k = 0
First Set
Follow Set
? k = 1
First Set
Follow Set
o k = 0
First Set
Follow Set
- BEGIN(0)
Follow Set
BEGIN k = 0
First Set
Follow Set
o k = 0
First Set
Follow Set
sequence_of_statements k = 0
First Set
Follow Set
? k = 1
First Set
Follow Set
o k = 0
First Set
Follow Set
- EXCEPTION(0)
Follow Set
EXCEPTION k = 0
First Set
Follow Set
* k = 1
First Set
Follow Set
exception_handler k = 0
First Set
Follow Set
exception_handler(0)
o k = 0
First Set
Follow Set
END k = 0
First Set
Follow Set

```



```

Follow Set
IS k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
declarative_part k = Ø ref = 3
First Set
Follow Set
o k = Ø
First Set
Follow Set
BEGIN k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
sequence_of_statements k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
? k = 1
First Set
Follow Set
o k = Ø
First Set
Follow Set
- EXCEPTION(Ø)
Follow Set
EXCEPTION k = Ø
First Set
Follow Set
* k = 1
First Set
Follow Set
exception_handler k = Ø
First Set
Follow Set
- exception_handler(Ø)
Follow Set
o k = Ø
First Set
Follow Set
END k = Ø
First Set
Follow Set
o k = Ø
First Set
Follow Set
? k = 1
First Set
Follow Set
designator k = Ø
First Set
Follow Set
- designator(Ø)
Follow Set
SEMICOLON k = Ø

```

```

!! !! !! !! !! First Set
!! !! !! !! !! Follow Set
!! !! !! !! !! SEMICOLON k = 0
!! !! !! !! !! First Set
!! !! !! !! !! - SEMICOLON(0)
!! !! !! !! !! Follow Set
!! !! !! !! !! empty k = 0
!! !! !! !! !! First Set
!! !! !! !! !! Follow Set

```

\*\*\*\*\*

```

package_declaration
! package_specification k = 0 ref = 1
! First Set
! Follow Set

```

\*\*\*\*\*

```

With_clause
! o k = 0
! First Set
! Follow Set
! WITH k = 0
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! UNIT_name k = 0 ref = 1
! First Set
! Follow Set
! * k = 1
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! COMMA(0)
! Follow Set
! COMMA k = 0
! First Set
! Follow Set
! UNIT_name k = 0 ref = 2
! First Set
! Follow Set

```

\*\*\*\*\*

```

use_clause
! o k = 0
! First Set
! Follow Set
! USE k = 0
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! PACKAGE_name k = 0 ref = 1
! First Set
! Follow Set

```



```

declarative_item
!! k = 1
!! First Set
!! Follow Set
!! declaration k = Ø
!! First Set
!! - declaration(Ø)
!! Follow Set
!! use_clause k = Ø ref = 2
!! First Set
!! - USE(Ø)
!! Follow Set

```

```

declarative_part
! o k = Ø
! First Set
! Follow Set
! * k = 1
! First Set
! Follow Set
! ! declarative_item k = Ø ref = 1
! ! First Set
! ! - USE(Ø)
! ! - declaration(Ø)
! ! Follow Set
! ! * k = 1
! ! First Set
! ! Follow Set
! ! program_component k = Ø ref = 1
! ! First Set
! ! - FUNCTION(Ø)
! ! - PROCEDURE(Ø)
! ! - PACKAGE(Ø)
! ! Follow Set
! !

```

**program\_component**  
**i | k = 2**

```

!! First Set
!! Follow Set
!! body k = Ø ref = 1
!!
!! First Set
!! - FUNCTION(Ø) designator(Ø)
!! - PROCEDURE(Ø) Identifier(Ø)
!! - PACKAGE(Ø) BODY(Ø)
!! Follow Set
!! package_declaration k = Ø ref = 2
!! First Set
!! - PACKAGE(Ø) Identifier(Ø)
!! Follow Set

*****

formal_part
! o k = Ø
!! First Set
!! Follow Set
!! LEFT_PAREN k = Ø
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! parameter_declaration k = Ø ref = 1
!! First Set
!! Follow Set
!! * k = 1
!! First Set
!! Follow Set
!! o k = Ø
!! Follow Set
!! First Set
!! - SEMICOLON(Ø)
!! Follow Set
!! SEMICOLON k = Ø
!! First Set
!! Follow Set
!! parameter_declaration k = Ø ref = 2
!! First Set
!! Follow Set
!! RIGHT_PAREN k = Ø
!! First Set
!! Follow Set

*****

parameter_declaration
! o k = Ø
!! First Set
!! Follow Set
!! Identifier_list k = Ø ref = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set

```

```

!! Follow Set
!! mode k = Ø ref = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! subtype_indication k = Ø ref = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! ? k = 1
!! First Set
!! Follow Set
!! COLON_EQUALS k = Ø
!! First Set
!! - COLON_EQUALS(Ø)
!! Follow Set
!! expression k = Ø
!! First Set
!! Follow Set

```

\*\*\*\*\*

```

Identifier_list
!! o k = Ø
!! First Set
!! Follow Set
!! Identifier k = Ø
!! First Set
!! Follow Set
!! * k = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! - COMMA(Ø)
!! Follow Set
!! COMMA k = Ø
!! First Set
!! Follow Set
!! Identifier k = Ø
!! First Set
!! Follow Set

```

\*\*\*\*\*

```

mode
!! k = 2
!! First Set
!! Follow Set
!! ? k = 1
!! First Set
!! - Identifier(1) LEFT_PAREN(Ø)
!! - Identifier(1) DOT(Ø)
!! - Identifier(1) constraint(Ø)
!! - Identifier(1) COLON_EQUALS(Ø)

```

新刊

— 10 —

```

!! Follow Set
!! empty k = Ø
!! First Set
!! Follow Set
!! name_1 k = Ø ref = 1
!! First Set
!! Follow Set

```

\*\*\*\*\*

```

name_1
!! k = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! - LEFT_PAREN(Ø)
!! - DOT(Ø)
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! empty k = Ø
!! First Set
!! Follow Set
!! k = 2
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! - LEFT_PAREN(Ø) expression(Ø)
!! Follow Set
!! LEFT_PAREN k = Ø
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! expression k = Ø
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! * k = 1
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! - COMMA(Ø)
!! Follow Set
!! COMMA k = Ø
!! First Set
!! Follow Set
!! expression k = Ø
!! First Set
!! Follow Set
!! RIGHT_PAREN k = Ø
!! First Set

```

```

! ! ! ! ! Follow Set
! ! ! ! ! k = 2
! ! ! ! ! First Set
! ! ! ! ! - LEFT_PAREN(0) discrete_range(0)
! ! ! ! ! - DOT(0) identifier(0)
! ! ! ! ! - DOT(0) ALL(0)
! ! ! ! ! - DOT(0) operator_symbol(0)
! ! ! ! ! - LEFT_PAREN(0) parameter_association(0)
! ! ! ! ! - LEFT_PAREN(0) RIGHT_PAREN(0)
! ! ! ! ! Follow Set
! ! ! ! ! o k = 0
! ! ! ! ! First Set
! ! ! ! ! - LEFT_PAREN(0) discrete_range(0)
! ! ! ! ! Follow Set
! ! ! ! ! LEFT_PAREN k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! o k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! discrete_range k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! RIGHT_PAREN k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! k = 2
! ! ! ! ! First Set
! ! ! ! ! - DOT(0) identifier(0)
! ! ! ! ! - DOT(0) ALL(0)
! ! ! ! ! - DOT(0) operator_symbol(0)
! ! ! ! ! - LEFT_PAREN(0) parameter_association(0)
! ! ! ! ! - LEFT_PAREN(0) RIGHT_PAREN(0)
! ! ! ! ! Follow Set
! ! ! ! ! o k = 0
! ! ! ! ! First Set
! ! ! ! ! - DOT(0) identifier(0)
! ! ! ! ! Follow Set
! ! ! ! ! DOT k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! identifier k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! k = 2
! ! ! ! ! First Set
! ! ! ! ! - DOT(0) ALL(0)
! ! ! ! ! - DOT(0) operator_symbol(0)
! ! ! ! ! - LEFT_PAREN(0) parameter_association(0)
! ! ! ! ! - LEFT_PAREN(0) RIGHT_PAREN(0)
! ! ! ! ! Follow Set
! ! ! ! ! o k = 0
! ! ! ! ! First Set
! ! ! ! ! - DOT(0) ALL(0)
! ! ! ! ! Follow Set
! ! ! ! ! DOT k = 0
! ! ! ! ! First Set
! ! ! ! ! Follow Set
! ! ! ! ! ALL k = 0

```

```

First Set
Follow Set
k = 1
First Set
- DOT(Ø) operator_symbol(Ø)
- LEFT_PAREN(Ø) parameter_association(Ø)
- LEFT_PAREN(Ø) RIGHT_PAREN(Ø)
Follow Set
k = Ø
o First Set
- DOT(Ø)
Follow Set
DOT k = Ø
First Set
Follow Set
operator_symbol k = Ø
First Set
Follow Set
k = Ø
First Set
- LEFT_PAREN(Ø)
Follow Set
LEFT_PAREN k = Ø
First Set
Follow Set
k = Ø
o First Set
Follow Set
? k = 1
First Set
Follow Set
o k = Ø
First Set
- parameter_association(Ø)
Follow Set
parameter_association k = Ø
First Set
Follow Set
* k = 1
First Set
Follow Set
o k = Ø
First Set
- COMMA(Ø)
Follow Set
COMMA k = Ø
First Set
Follow Set
parameter_association k = Ø
First Set
Follow Set
RIGHT_PAREN k = Ø
First Set
Follow Set
k = Ø
o First Set
Follow Set
empty k = Ø
First Set

```

```

!! Follow Set
!! name_1 k = Ø ref = 2
!! First Set
!! Follow Set
!! empty k = Ø
!! First Set
!! - END(2)
!! - PRIVATE(2)
!! - declaration(2)
!! - BEGIN(2)
!! - constraint(2)
!! - SEMICOLON(2)
!! - IS(2)
!! - COLON_EQUALS(2)
!! - expression(2)
!! - USE(2)
!! - FUNCTION(2)
!! - PROCEDURE(2)
!! - PACKAGE(2)
!! - WITH(2)
!! - COMMA(2)
!! - END(1)
!! - BEGIN(1)
!! - PRIVATE(1)
!! - declaration(1)
!! - constraint(1)
!! - COLON_EQUALS(1)
!! - expression(1)
!! - SEMICOLON(1)
!! - IS(1)
!! - USE(1)
!! - FUNCTION(1)
!! - PROCEDURE(1)
!! - PACKAGE(1)
!! - WITH(1)
!! - COMMA(1)
!! Follow Set

```

\*\*\*\*\*

```

package_specification
! o k = Ø
!! First Set
!! Follow Set
!! PACKAGE k = Ø
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! Identifier k = Ø
!! First Set
!! Follow Set
!! o k = Ø
!! First Set
!! Follow Set
!! IS k = Ø
!! First Set
!! Follow Set

```



```

!! o k = 0
!! First Set
!! Follow Set
!! * k = 1
!! First Set
!! Follow Set
!! declarative_item k = 0 ref = 2
!! First Set
!! - USE(0)
!! - declaration(0)
!! Follow Set
!! o k = 0
!! First Set
!! Follow Set
!! ? k = 1
!! First Set
!! Follow Set
!! o k = 0
!! First Set
!! - PRIVATE(0)
!! Follow Set
!! PRIVATE k = 0
!! First Set
!! Follow Set
!! * k = 1
!! First Set
!! Follow Set
!! declarative_item k = 0 ref = 3
!! First Set
!! - USE(0)
!! - declaration(0)
!! Follow Set
!! o k = 0
!! First Set
!! Follow Set
!! END k = 0
!! First Set
!! Follow Set
!! ? k = 1
!! First Set
!! Follow Set
!! Identifier k = 0
!! First Set
!! - Identifier(0)
!! Follow Set

```

\*\*\*\*\*

```

PACKAGE_name
! name k = 0 ref = 1
! First Set
! Follow Set

```

\*\*\*\*\*

```

subtype_indication
! o k = 0
! First Set
! Follow Set

```

```

! type_mark k = 0 ref = 1
! First Set
! Follow Set
! ? k = 1
! First Set
! Follow Set
! ! constraint k = 0
! First Set
! - constraint(0)
! Follow Set

*****

subprogram_specification
! k = 1
! First Set
! Follow Set
! o k = 0
! First Set
! - PROCEDURE(0)
! Follow Set
! ! PROCEDURE k = 0
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! ! Identifier k = 0
! First Set
! Follow Set
! ? k = 1
! First Set
! Follow Set
! ! formal_part k = 0 ref = 1
! First Set
! - LEFT_PAREN(0)
! Follow Set
! o k = 0
! First Set
! - FUNCTION(0)
! Follow Set
! ! FUNCTION k = 0
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! ! designator k = 0
! First Set
! Follow Set
! o k = 0
! First Set
! Follow Set
! ! ? k = 1
! First Set
! Follow Set
! ! formal_part k = 0 ref = 2
! First Set
! - LEFT_PAREN(0)

```

```

!! !! !! !! !! Follow Set
!! !! !! !! !! k = 0
!! !! !! !! !! First Set
!! !! !! !! !! Follow Set
!! !! !! !! !! RETURN k = 0
!! !! !! !! !! First Set
!! !! !! !! !! Follow Set
!! !! !! !! !! subtype_indication k = 0 ref = 2
!! !! !! !! !! First Set
!! !! !! !! !! Follow Set
*****

```

```

type_mark
! TYPE_name k = 0 ref = 1
! First Set
! Follow Set
*****

```

```

TYPE_name
! name k = 0 ref = 2
! First Set
! Follow Set
*****

```

```

UNIT_name
! name k = 0 ref = 3
! First Set
! Follow Set
*****

```

END PRODUCTION TREES

## Appendix J

### LL User's Manual

George B. Paprotny, Capt, USAF  
December, 1983

#### Introduction

LL provides a tool for creating a parser and lexical analyzer for any LL(k) grammar. The user provides a parser specification which includes the tokens used, the rules describing the lexical analyzer, the grammar rules for the parser, and a set of code which is user-unique. The lexical and grammar rules may have actions, which are also user-specified. LL then determines the value of k, and creates the output parser.

LL is written in C, and creates a parser/lexical analyzer also written in C. In addition, many of LL's rules follow the syntax rules of C.

The terminology used in this manual is standard. Any textbook on compilers may be referenced if confusion arises, however, a complete reference to the terminology and theory behind LL may be found in the author's thesis.

#### General Format

The general format of an LL input specification is:

```
token declarations
%%
lexical rules
%%
```

production rules

%%

variable and subroutine declarations

The lexical rules section may be empty. Also, the variable and subroutine declarations section may be omitted, and if it is, the %% immediately preceding it is optional. If both are omitted, the format would be

token declarations

%%

%%

production rules

LL has the capability to trace its progress through an input grammar, and report that progress on the standard output. To turn this on, the token declarations must be preceded by a line which has "%TRACE" beginning in column 1. No token declarations may be on this line.

Throughout the rules, tokens and non-terminals may be of arbitrary length. They consist of an alphabetic character, followed by a sequence of digits, underscores, and alphabetic characters. In addition, non-terminals may be surrounded by "<" and ">".

### The Token Declarations

The token declarations section consists of a series of token names. Each token used in any part of the lexical analyzer and parser must be declared here. Blanks, tabs, and newlines are ignored in this section.

### The Lexical Rules Section

Each rule in the lexical rules section consists of a regular expression followed by an optional action. The rules must begin in column 1, and may continue for many lines. The regular expression is terminated by a blank, tab, or newline, and the action is a series of C statements. If the action continues on multiple lines, each successive line must begin with a blank, tab, or newline. When an action is not present, the characters recognized by the regular expression are ignored, and a new rule is begun.

Regular expressions in LL use the following operators:

x	-	the character "x"
"x"	-	"x", even if x is an operator or space
\x	-	"x", even if x is an operator or a space, unless it is one of the standard C codes, such as n, t, etc.
[xy]	-	x or y
[x-z]	-	any character between x and z, including x and z
[^xy]	-	any character except x or y
.	-	any character except newline
x?	-	optional x
x*	-	>= 0 instances of x
x+	-	>= 1 instances of x
x y	-	x or y
(x)	-	x

Note that parentheses may surround a sub-expression. For example, (ab)|(cd) is the same as ab|cd, but not the same as a(b|c)d.

For a token to be passed to the parser, a "return(token-name)" must be issued in the action for the regular expression that recognized the character string. If no return is issued, the action will be executed, and then the lexical analyzer

will begin reading the next token.

The lexical rules in a grammar may allow the same token to be recognized by two different lexical rules. If this is so, then the first rule will be the one whose action will be performed.

### The Production Rules

A production rule in LL has the form:

A : production-set ;

where A is a non-terminal, and production-set is a series of zero or more terminals, non-terminals, options, closures, closure-pluses, and actions.

An option is one entry in a production set, and is designated by [ production-set ]. This defines that the production set is optional. A closure is { production-set }, and defines zero or more occurrences of the production set. Finally, closure-plus is { production-set }+, and defines one or more occurrences of the production set.

Actions in a production set are delimited by ={ and }. They consist of a sequence of C statements that will be executed during the parse.

One last method of defining a production set is by surrounding it with parentheses, as in A : B ( C | D ) E ; .

When LL production rules are defined, one of the non-terminals is the start or goal symbol. By default, the start symbol is the non-terminal on the left-hand side of the first production rule.

### Errors in Production Rules

There are a host of errors which may occur in production rules, other than "standard" syntax and spelling errors.

These include the following:

a. Left Recursion. By definition, no LL(k) grammar may have a left resursive non-terminal. Thus, grammars of the form

$$\begin{array}{l} A : B a \mid c ; \\ B : A b \mid d ; \end{array}$$

are illegal.

b. Undefined names. Any names that are not declared as a token or non-terminal are considered undefined.

c. Unused productions. Only the first production rule may have its non-terminal unused by another production rule. Also, all non-terminals must be used by some sequence of references that started at the start symbol.

d. Closure contains empty. Option, closure and closure-plus production sets may not have the empty string as a member of their first sets.

Also, a grammar may not be LL(k) for any k. Most grammars which do not have one of the above problems are LL(k), but there are some which are not, and the list of possible problems is endless. For example:

$$A : \{ a \} b \mid \{ a \} c ;$$

is not LL(k), because determining which alternate to pursue requires an infinite amount of lookahead.



## Values

LL provides the user with the capability of giving each item in a production set a "value". For tokens, this value is a unique integer for each different token, and for non-terminals, the value is the value of the non-terminal's production rule. Actions may also have values.

The values are referenced in the actions by using \$1, \$2, etc. For example, in:

A : B C D = { action } ;

the value of B is \$1, C is \$2, and D is \$3. (The \$n values are used as variables.) The action is \$4. The value of A upon completion of the rule is the value of the last item in the production set, which in this case is the action. Actions have an intrinsic value of 0, but the value may be changed by setting \$\$\$. In the previous example, if the user wanted to preserve the value of C, the statement \$\$ = \$2; would be inserted in the action.

Each production set within a rule is numbered separately for the purpose of using values. For example, in

A : B { C D } [ E F ] ( G { H I }+ ) ;

{ C D } is \$2, [ E F ] is \$3, and ( G { H I }+ ) is \$4. C and D are \$1 and \$2 respectively for the production set within the braces. Because {} defines a set of zero or more, there may be many values for \$2. Each of these values may be referenced using the C array construct, i.e. \$2[i]. Here, \$2[0] is the number of occurrences, and \$2[1] through \$2[n]

are the values of the occurrences. (This applied to [] and {}+ as well, except that [] will only have zero or one element.)

For the example in the above paragraph, with the input

B E F G H I H I

the values for each nested production set would be

```

$1      = B
$2[0]   = 0
$3[0]   = 1
  $1     = E
  $2     = F
  $$     = F
$3[1]   = F
  $1     = G
  $2[0]  = 2
    $1   = H (first)
    $2   = I (first)
    $$   = I (first)
  $2[1]  = I (first)
    $1   = H (second)
    $2   = I (second)
    $$   = I (second)
  $2[2]  = I (second)
    $$   = I (second)
$4       = I (second)
$$       = I (second)

```

Note that the value of \$4 in the outer production set is the value of the last item in the array for the closure.

### The LL Environment

LL has a host of output files, some temporary and some permanent. All the files have names that begin with "LL.", so the user is warned not to keep any files that begin with those characters in the working directory while LL is running. The output file LL.c is the parser, which must be compiled by the user. The file LL.output contains straightforward graph-

ical descriptions of the finite state machine and production trees created from the lexical and production rules. The standard output is not used, unless %TRACE is specified. In that case, LL's progress is printed there. Also, the standard error is used only when an error occurs. Any detailed descriptions of the error can be found on the standard output.

The standard input is where the input grammar is taken from. There is one option for LL. If a grammar is known to be LL(n), the value of n may be input on the command line.

The output parser in LL.c has the entry point "compile", which is a subroutine with no arguments. If a syntax error occurs, the parser will stop, and an error message will print. Under normal circumstances, the parser will print (on the standard output) each input line as it receives it from the standard input.

The use of values in actions has already been discussed, but LL also provides the string that made up the last parsed token in the character array "LL\_text", and that token's length in "LL\_length".

#### User-Supplied Lexical Analyzer

If no lexical rules are given, no lexical analyzer will be created. In this case, the interfaces to the parser must be taken into consideration. LL expects the lexical analyzer routine to be called "LL\_lex", for it to have no arguments, and for it to return an integer defining which tokens are

found. It also expects the string of characters that make up the returned token to be placed in a character array called "LL\_area". This variable must be explicitly declared in the variable and subroutined declarations, and the LL\_lex function must also be supplied.

LEX is another UNIX-supplied lexical analyzer creator. LL will interface to LEX if the following statements are included in the declarations:

```
# define LL_lex yylex
# define LL_area yytext
extern char LL_area[];
```

If no lexical analyzer is declared, LL will anticipate that the user needs a list of defines for the tokens. This list will be placed in the output file "LL.h", and is suitable for inclusion in LEX, or any user-supplied lexical analyzer. This file will also contain the defined name END\_OF\_FILE, which has the value 0 (zero). Zero must be returned from the lexical analyzer when it finds the end of file.

#### Re-Definition of Internal Maximums

There are two limitations within LL that may be changed by the user. First, a token received may be of any size up to 200 characters. This maximum size is defined by the defined name LL\_MAXCHAR. It may be redefined to any size by the user. Also, there is a maximum number of \$n items that can be saved during a parse. This value is 10,000. There

There are also 10,000 possible entries for the various values in closures. If either one of these overflows, LL will abort with a value stack overflow message. They may be redefined by defining the name LL\_VAL\_STACK.

### Bugs

LL has three problems. These are:

a. It uses too much memory for large grammars that have  $k > 2$  or 3, and may go beyond the 6 MB limit set by UNIX.

Smaller grammars can be almost any  $k$  and still be completed.

b. Some grammars which are LL( $k$ ) but not strong LL( $k$ ) may not be recognized as LL( $k$ ) by LL. For example, the following are two grammars for an LL(2) language:

A : aBbc   Bc ;	A : aCbc   Cc ;
B : b   c   ;	C : B ;
	B : b   c   ;

The right grammar is equivalent to the left one, but the right one will not process correctly. This problem occurs because the parser must have the follow set of B to determine which alternative in B to take. On the left, the two possible follow sets are well defined, and can be used to decide which path to take. LL can not look back more than one production to find the appropriate follow sets, so the one on the right will be an error. (This error occurs very infrequently, but is difficult to find when it does.)

c. The value processor will accept only variable names or integers between the brackets for a  $\$n[]$  construction.

Thus, expressions of any sort are not allowed.

Acknowledgement

The material reported herein is based on the author's thesis: LL - A Generator of Recursive Descent Parsers for LL(k) Languages, submitted in partial fulfillment of the requirements for the Master of Science Degree at the Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, in December 1983.

## Appendix K

### Code for LL

The code for LL was created in many separate files, and each was compiled separately. Each file begins on a new page in this listing. The first file is a C "include" file, and contains the definitions of all data structures, and some global variables. The second file is the entry point to LL.

The last five files are marked "Standard Code File", and are used by the "finish\_write" routine during the final output.

```

/* struct.h */

/* list of system include files that are needed */
#include "stdio.h"
#include "sys/types.h"
#include "sys/dir.h"
#include "sys/stat.h"

#define MAX_ITEMS 100 /* max number of items in any concat list */
#define MAX_CHAR_SET 128 /* number of characters in character set */
#define TRUE 1
#define FALSE 0
#define WRITE "w" /* used in opening files */
#define READ "r" /* used in opening files */
#define EMPTY -1

/* list of node types */
#define closure 1
#define plus 2
#define alternation 3
#define concat 4
#define option 5
#define non_terminal 6
#define terminal 7
#define empty 8

/* list of file names for all files */
#define ll_output "LL.output"
#define ll_c "LL.c"
#define tmp_lexaction "LL.Tmp.lexac"
#define tmpprodaction "LL.Tmp.prdac"
#define tmpvarsub "LL.Tmp.varsb"
#define tmptokendef "LL.h"
#define tmpparser "LL.Tmp.parsr"
#define tmptranstable "LL.Tmp.trans"

/* list of file names for standard code files */
#define code_file_1 "z1.c"
#define code_file_2 "z2.c"
#define code_file_3 "z3.c"
#define code_file_4 "z4.c"
#define code_file_5 "z5.c"

/* defines for all file control blocks in the system */
#ifdef FDEF_HERE
extern /* used to insure only one program has them defined.
currently defined in "files.c" */
#endif
FILE *ll_c, *output, *tmp_lex_action, *tmp_prod_action, *var_sub_code,
      *token_def_code, *tmp_parser, *tmp_trans_table;

/* when set by lex, will cause a trace of LL's execution */
extern int trace;

struct FSA /* states */
{
    struct FSA *next_state;
    int state_num;
    int action_id;
};

```



```

    struct trans_type      *transition[MAX_CHAR_SET];
    struct trans_type      *EMPTY_transition;
};

struct trans_type /* transitions */
{
    struct trans_type      *next_trans;
    struct FSA             *to_state;
};

struct production_tree /* production tree head */
{
    char                   *nterm_name;
    struct tree_node       *tree_head;
    struct production_tree *next_tree;
    int                    ptree_mark;
};

struct tree_node /* production tree internal node */
{
    int                    K;
    struct first_follow    *first_set;
    struct first_follow    *follow_set;
    int                    action_id;
    int                    node_type;
    union
    {
        struct tree_node  *left_child; /* !,.,
        struct tree_node  *child;      /* *,?,
        struct production_tree *non_term; /* non_terminal*/
        struct term_list  *term_ptr;  /* terminal */
    }
    left;

    union
    {
        struct tree_node  *right_child; /* !,.,
        int                ref_num;      /* non_terminal*/
    }
    right;
};

struct term_list /* terminal definition nodes */
{
    char
    int
    struct term_list
};

struct first_follow /* a list of these defines a first set.
one is the front of a first item */
{
    struct token
    struct first_follow
};

struct token /* a list of these defines a first item.
one is an instance of a token */
{
    struct term_list
    struct term_list
};

```

\*next\_token;  
ref\_number;

struct token  
int  
);

```

# include "struct.h"

/*
standard input - definition
standard output - system messages and error messages
                  if the standard input contains "%TRACE" as the first line,
                  the processes performed will be traced.
standard error - final error message
LL.output      - interesting user output
LL.c           - compiler
*/

/*
NAME : main
PURPOSE : to control the running of LL
CALLED BY : n/a
CALLS : decorate, make_parser, open_files, yacc, make_lexical_analyzer,
        finish_write, delete_temp_files
ARGUMENTS PASSED IN : optional maximum k value
VALUE RETURNED : n/a
LOCAL VARIABLES :
    max_K - integer value of maximum k input, if none input, = 0
    ndfsa - pointer to the FSA created
    trees - pointer to production trees
    k - temporary storage value of k for grammar
    lex_made - boolean flag for whether the lexical analyzer is to be
              output
*/

main(argc,argv)
int argc;
char *argv[];
{
    struct FSA *ndfsa;
    struct production_tree *trees,*decorate();
    int max_K, k, make_parser();
    int lex_made;
    debug("begin main()",1);

    max_K = 0;
    if (argc == 2)
        while (*argv[1] != '\0')
            max_K = (max_K * 10) + (*argv[1]++ - '0');

    open_files();
    yacc(&ndfsa,&trees);
    if (trace)
        printf("Creating lexical analyzer.\n"),fflush(stdout);

    lex_made = make_lexical_analyzer(ndfsa);
    if (trace)
        printf("Decorating production trees.\n"),fflush(stdout);

    decorate(trees,max_K);
    output_prods(trees);
    if (trace)
        printf("Creating parser.\n"),fflush(stdout);

    k = make_parser(trees);
    if (trace)

```

```

printf("Writing final parser.\n"), fflush(stdout);

finish_write(lex_made);
fflush(stdout);
delete_temp_files(lex_made);

if (trace)
{
    printf("\nk = %d\n\nEnd LL(k) Compiler Generator\n", k);
    fflush(stdout);
};
debug("end    main()", 1);
exit(0);
}

```

```

# include "struct.h"
/*****
/*
NAME : convert_NDFSA_DFSA
PURPOSE : control the process of converting NDFSA to DFSA
CALLED BY : make_lexical_analyzer
CALLS : remove_empty_transitions, make_deterministic, delete_unreachable
ARGUMENTS PASSED IN : pointer to the NDFSA
VALUE RETURNED : none
LOCAL VARIABLES :
    fsa - temporary pointer to FSA
    snum - used for numbering states after removing unreachable states
*/
convert_NDFSA_DFSA (ndfsa)
struct FSA *ndfsa;
{
    struct FSA *fsa;
    int snum;
    debug("begin convert_NDFSA_DFSA ()",1);

    snum = 0;
    if (trace)
        printf(" Removing empty transitions.\n"),fflush(stdout);

    remove_empty_transitions(ndfsa);
    if (trace)
        printf(" Making fsa deterministic.\n"),fflush(stdout);

    make_deterministic(ndfsa);
    if (trace)
        printf(" Deleting unreachable states.\n"),fflush(stdout);

    delete_unreachable(ndfsa);
    /* number the states */
    for(fsa=ndfsa;fsa!=NULL;fsa=fsa->state_num = snum++,fsa=fsa->next_state);
    debug("end convert_NDFSA_DFSA ()",1);
}
/*****
/*
NAME : remove_empty_transitions
PURPOSE : to remove empty transitions from the NDFSA
CALLED BY : convert_NDFSA_DFSA
CALLS : delete_transition, merge_into
ARGUMENTS PASSED IN : pointer to NDFSA
VALUE RETURNED : none
LOCAL VARIABLES :
    fsa - temporary pointer to a state
    to - maintains the state an empty transition points to while
        it is being deleted
    FLAG - set to 1 if an EMPTY transition is discovered
*/
static
remove_empty_transitions(ndfsa)

```

```

        || ( into_state->action_id > from_state->action_id ) )
        into_state->action_id = from_state->action_id;
    }
    debug("end merge_into()",1);
}

/*****
*/
NAME : make_deterministic
PURPOSE : remove all multiple transitions on the same character
CALLED BY : convert_NDFSA_DFSA
CALLS : new_state, merge_into, change_trans, modify_state_set
ARGUMENTS PASSED IN : pointer to NDFSA
VALUE RETURNED : none
LOCAL VARIABLES :
    tr - pointer to a pointer to a transition
    t - pointer to a transition
    s - pointer to a transition
*/

static
make_deterministic(ndfsa)
struct FSA *ndfsa;
{
    struct trans_type **tr,*t,*s;
    int FLAG,i;
    struct FSA *state,*laststate,*temp,*new_state();
    debug("begin make_deterministic()",1);
    for (laststate=ndfsa;
        laststate->next_state!=NULL;
        laststate=laststate->next_state);
    for (;ndfsa!=NULL;ndfsa=ndfsa->next_state)
    {
        for (i=EMPTY;i<MAX_CHAR_SET;i++)
        {
            switch(i)
            {
                case EMPTY:tr= &ndfsa->EMPTY_transition;break;
                default : tr= &ndfsa->transition[i];break;
            };
            if (*tr != NULL)
            {
                if ((*tr)->next_trans != NULL)
                {
                    state=laststate=new_state(laststate);
                    for (t= *tr;t!=NULL;t=t->next_trans)
                    {
                        merge_into(state,t->to_state);
                        t= *tr;
                        *tr=NULL;
                        change_trans(t,ndfsa,state);
                        *tr=t;
                        modify_state_set(t,state);
                    };
                };
            };
        };
    };
    debug("end make_deterministic()",1);
}

```

```

/*****

```

```

/*
NAME : change_trans
PURPOSE : to accept a state set, and for each set of transitions in the
FSA, if the state set is the same, change the transitions to point
to the new state
CALLED BY : make_deterministic
CALLS : compare_state_sets, modify_state_set
ARGUMENTS PASSED IN :
    pointer to a set of transitions that are for a state
    pointer to the NDFSA
    pointer to a new state being created
VALUE RETURNED : none
LOCAL VARIABLES :
    i - Integer temporary
*/

```

```

static
change_trans(state_set, fsa, new_state)
struct trans_type *state_set;
struct FSA *fsa, *new_state;
{
    int i;
    debug("begin change_trans()", 1);
    for (ifsa := NULL; fsa = fsa->next_state)
    {
        if (compare_state_sets(state_set, fsa->EMPTY_transition))
            modify_state_set(fsa->EMPTY_transition, new_state);
        for (i=1; i<MAX_CHAR_SET; i++)
            if (compare_state_sets(state_set, fsa->transition[i]))
                modify_state_set(fsa->transition[i], new_state);
    }
    debug("end change_trans()", 1);
}

```

```

/*****

```

```

/*
NAME : modify_state_set
PURPOSE : change one set of transitions to a single transition
CALLED BY : change_trans, make_deterministic
CALLS : free
ARGUMENTS PASSED IN :
    pointer to a transition
    pointer to a new state that will replace the set of states
in the transition
VALUE RETURNED : none
LOCAL VARIABLES :
    t - temporary pointer to a transition
*/

```

```

static
modify_state_set(old_set, new_state)
struct trans_type *old_set;
struct FSA *new_state;
{
    struct trans_type *t;
    debug("begin modify_state_set()", 1);
}

```

```

old_set->to_state = new_state;
while (old_set->next_trans != NULL)
{
    t=old_set->next_trans;
    old_set->next_trans = t->next_trans;
    free(t);
};
debug("end    modify_state_set()",1);
}

/*****
/*
NAME : compare_state_sets
PURPOSE : compare two transition sets, and return TRUE or FALSE compare
CALLED BY : change_trans
CALLS :
ARGUMENTS PASSED IN : pointers to two transition sets
VALUE RETURNED : TRUE or FALSE
LOCAL VARIABLES :
t - loop variable to pointer to transition
FLAG - boolean to keep track of whether the transitions are =
*/

static int
compare_state_sets(set1,set2)
struct trans_type *set1,*set2;
{
    struct trans_type *t;
    int FLAG;
    for (;set1!=NULL;set1=set1->next_trans)
    {
        FLAG = FALSE;
        for (t=set2;t!=NULL;t=t->next_trans)
        {
            if (set1->to_state == t->to_state)
                FLAG = TRUE;
        };
        if (!FLAG)
        {
            return(0);
        };
    };
    return(1);
}

/*****
/*
NAME : delete_unreachable
PURPOSE : return unreachable states to free space
CALLED BY : convert_NDFSA_DFSA
CALLS : free
ARGUMENTS PASSED IN : pointer to NDFSA
VALUE RETURNED : none
LOCAL VARIABLES :
t,s - temporary pointers to transitions
pointer to a state
i - integer to loop through possible transitions

```



```

*/      FLAG - boolean to keep track of whether any state is unreachable

static
delete_unreachable(ndfsa)
struct FSA *ndfsa;
{
    struct trans_type *t,*s;
    struct FSA *fsa;
    int i,FLAG;
    debug("begin delete_unreachable()",1);
    FLAG = TRUE;
    for (fsa=ndfsa;fsa!=NULL;fsa->state_num=FALSE,fsa=fsa->next_state);
    ndfsa->state_num=TRUE;
    while (FLAG)
    {
        FLAG = FALSE;
        for (fsa=ndfsa;fsa!=NULL;fsa=fsa->next_state)
        {
            if (fsa->state_num)
            {
                for (i=EMPTY;i<MAX_CHAR_SET;i++)
                {
                    switch(i)
                    {
                        case EMPTY:t=fsa->EMPTY_transition;break;
                        default: t=fsa->transition[i];break;
                    };
                    for (;t!=NULL;t=t->next_trans)
                    {
                        if(t->to_state->state_num == FALSE)
                        {
                            FLAG = TRUE;
                            t->to_state->state_num = TRUE;
                        };
                    };
                };
            };
        };
        for (fsa=ndfsa->next_state;fsa!=NULL;)
        {
            if (fsa->state_num == FALSE)
            {
                ndfsa->next_state=fsa->next_state;
                for (i=EMPTY;i<MAX_CHAR_SET;i++)
                {
                    switch(i)
                    {
                        case EMPTY:t=fsa->EMPTY_transition;break;
                        default:t=fsa->transition[i];break;
                    };
                    while (t!=NULL)
                    {
                        s = t;
                        t=t->next_trans;
                        free(s);
                    };
                };
            };
        };
    };
}

```

```
free(fsa);
fsa = ndfsa->next_state;
}
else {
    ndfsa = fsa;
    fsa = fsa->next_state;
};
};
debug("end delete_unreachable()",1);
}
```

```

# include "struct.h"

static int DEBUG = FALSE;
/*****

/*
NAME : debug_ON
PURPOSE : to turn on debug capability
CALLED BY : n/a
CALLS :
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

debug_ON()
{
    DEBUG = TRUE;
}

/*****/

/*
NAME : debug_OFF
PURPOSE : to turn debug capability off
CALLED BY : n/a
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

debug_OFF()
{
    DEBUG = FALSE;
}

/*****/

/*
NAME : debug
PURPOSE : to print data on the standard output if the debug flag is on. prints
a return only if the 'retrn' flag is set on input.
CALLED BY : n/a
CALLS : none
ARGUMENTS PASSED IN :
character string to print
boolean to show whether to print a return or not
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

debug(outstr,retrn)
char *outstr;
int retrn;
{
    if (DEBUG)
    {

```

```

        printf("%s", outstr);
        if (retrn) printf("\n");
        fflush(stdout);
    };

}

/*****
/*
NAME : debug_char
PURPOSE : prints a single character on the standard input
CALLED BY : n/a
CALLS : none
ARGUMENTS PASSED IN : character
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
debug_char(c)
char c;
{
    if (DEBUG)
    {
        switch(c)
        {
            case EMPTY: printf("EMPTY"); break;
            default:
                printf("%x", c);
                break;
        };
    };
}

/*****
/*
NAME : debug_num
PURPOSE : to print a number on the standard input iff the debug flag is on
CALLED BY : n/a
CALLS : none
ARGUMENTS PASSED IN : integer to print
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
debug_num(num)
int num;
{
    if (DEBUG)
    {
        printf("%d", num);
    };
}

/*****
/*
NAME : print_token_list
PURPOSE : to print the token list on the standard output

```

```

CALLED BY : n/a
CALLS : none
ARGUMENTS PASSED IN : pointer to head of token list
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

```

```

print_token_list(head)
struct term_list *head;
{
    if (DEBUG)
    {
        debug("begin print_token_list()",1);
        printf("\nTOKEN LIST\n\n");
        for (;head!=NULL;head=head->next_term)
        {
            printf("%s\t",head->term_name);
            if (strlen(head->term_name) < 8)printf("\t");
            if (strlen(head->term_name) < 16)printf("\t");
            printf("%4d\n",head->term_num);
        };
        printf("\nEND TOKEN LIST\n");
        debug("end print_token_list()",1);
    };
}

```

```

/*****

```

```

/*
NAME : print_fsa
PURPOSE : to print a FSA in acceptable format on the standard output
CALLED BY : n/a
CALLS : pchar
ARGUMENTS PASSED IN : pointer to FSA
VALUE RETURNED : none
LOCAL VARIABLES :
    t - pointer to transition inside loop
    i - integer loop variable
    INSIDE - boolean to FLAG whether a set of transitions that all
             have the same to-state are being processed
*/

```

```

print_fsa(fsa)
struct FSA *fsa;
{
    struct trans_type *t;
    int i,INSIDE;
    if (DEBUG)
    {
        debug("begin print_fsa()",1);
        printf("\nSTATE LIST\n\n");
        for (;fsa!=NULL;fsa=fsa->next_state)
        {
            printf("%4d ! %3d\n",fsa->state_num,fsa->action_id);
            if (fsa->EMPTY_transition != NULL)
            {
                printf("    EMPTY =");
                for (t=fsa->EMPTY_transition;
                    t != NULL;

```

```

        t = t->next_trans)
    {
        printf(" %d",t->to_state->state_num);
    };
    printf("\n");
    for (i=0; i<MAX_CHAR_SET; i++)
    {
        if (fsa->transition[i] != NULL)
        {
            printf(" %s",pchar(i));
            for (t=fsa->transition[i]; t != NULL; t=t->next_trans)
            {
                printf(" %d",t->to_state->state_num);
            };
            printf("\n");
        };
    };
    printf("\nEND STATE LIST\n");
    debug("end print_fsa()",1);
};

/*****
/*
NAME : print_prod
PURPOSE : prints a production tree
CALLED BY : n/a
CALLS : print_node
ARGUMENTS PASSED IN : pointer to the production tree
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
print_prod(tree)
struct production_tree *tree;
{
    if (DEBUG)
    {
        debug("begin print_prod()",1);
        printf("\nPRODUCTION TREES\n\n");
        for (; tree != NULL; tree = tree->next_tree)
        {
            printf("*****\n\n%s\n",tree->nterm_name);
            print_node(tree->tree_head,1);
            printf("\n");
        };
        printf("\nEND PRODUCTION TREES\n\n");
        debug("end print_prod()",1);
    };
}

/*****
/*
NAME : print_node
PURPOSE : to print a single node of a tree

```

```

CALLED BY : print_prod
CALLS : print_node, print_f_set
ARGUMENTS PASSED IN :
    pointer to the sub-tree being processed
    the current indentation of the print
VALUE RETURNED : none
LOCAL VARIABLES :
    c - character string to hold the character designation of current node
    i - integer loop variable
*/

print_node(tree, indent)
struct tree_node *tree;
int indent;
{
    char *c;
    int i;
    if (!DEBUG) return;
    if (tree == NULL) return;
    switch (tree->node_type)
    {
        case empty : c = "empty" ; break;
        case plus : c = "+" ; break;
        case closure : c = "*" ; break;
        case option : c = "?" ; break;
        case concat : c = "o" ; break;
        case alternation : c = "|" ; break;
        case non_terminal : c = tree->left.non_term->term_name; break;
        case terminal : c = tree->left.term_ptr->term_name; break;
    };
    for (i=0; i<indent; printf("!", i), i++);
    printf("%s k = %d", c, tree->K);
    if (tree->node_type == non_terminal)
        printf(" ref = %d\n", tree->right.ref_num);
    else
        printf("\n");
    for (i=0; i<indent; printf("!", i), i++);
    printf(" First Set\n");
    print_f_set(tree->first_set, indent);
    for (i=0; i<indent; printf("!", i), i++);
    printf(" Follow Set\n");
    print_f_set(tree->follow_set, indent);
    switch (tree->node_type)
    {
        case closure :
            case option :
                print_node(tree->left.child, indent+1); break;
            case alternation :
            case plus :
            case concat :
                print_node(tree->left.left_child, indent+1);
                print_node(tree->right.right_child, indent+1); break;
            default :
                break;
    };
}

/*****

```

```

/*
NAME : print_f_set
PURPOSE : to print a first/follow set for a node
CALLED BY : print_node
CALLS : none
ARGUMENTS PASSED IN :
    pointer to the first/follow set
    the current indentation of the print
VALUE RETURNED : none
LOCAL VARIABLES :
    i - Integer loop variable to perform indentation
    space - Integer variable to remember how many spaces indented
    t - pointer to a token to loop through tokens in one first/follow set
*/

print_f_set(f,indent)
struct first_follow *f;
int indent;
{
    int i,space;
    struct token *t;
    char num[10];
    if (!DEBUG) return;
    for (;f!= NULL;f = f->next_set)
    {
        for (i=0;i<indent;printf("! "),i++);
        printf(" ");
        space = indent*2+6;
        for (t=f->first_token;t!=NULL;t=t->next_token)
        {
            sprintf(num,"%d",t->ref_number);
            if (space + strlen(t->term_node->term_name)
                + strlen(num) + 3 > 79)
            {
                printf("\n");
                space = indent*2+6;
                for (i=0;i<space;printf(" "),i++);
            };
            printf("%s{Xs} ",
                t->term_node->term_name,num);
            space = space + strlen(t->term_node->term_name)
                + strlen(num) + 3;
        };
        printf("\n");
    }
}

```



AD-A138 061

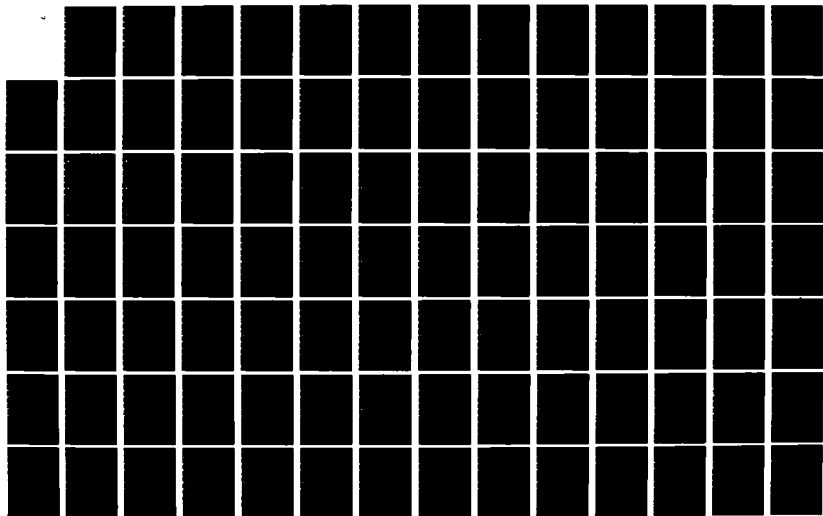
A GENERATOR OF RECURSIVE DESCENT PARSERS FOR LL(K)  
LANGUAGES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING G B PAPROTNY DEC 83  
AFIT/GCS/MA/83D-6

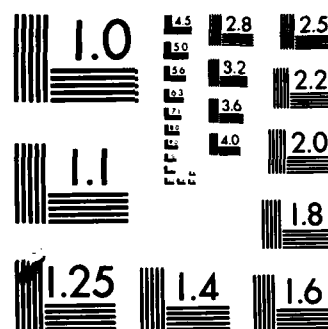
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

```

#include "struct.h"

/*
NAME : decorate
PURPOSE : to control the decoration of the production_trees
CALLED BY : main
CALLS : find_and_number_references, find_unused_productions,
        find_left_recursion, find_maximum_K, find_first_sets,
        find_follow_sets, find_adjusted_first_sets,
        find_K_for_nodes, remove_ff_sets
ARGUMENTS PASSED IN : pointer to the production trees; user input max-k, or 0
VALUE RETURNED : none
LOCAL VARIABLES :
        max_K - local definition of the maximum K
        NOT_LL - boolean that remembers whether the grammar is LL(k)
        for some k
        i - integer to loop between 1 and max_K
*/

decorate(trees, max_K_input)
struct production_tree *trees;
int max_K_input;
{
    int max_K, i, NOT_LL;
    debug("begin decorate()", 1);
    if (trace)
        printf(" Numbering non_terminal references.\n"), fflush(stdout);

    find_and_number_references(trees);
    if (trace)
        printf(" Finding undefined non_terminals.\n"), fflush(stdout);

    find_undefined_nonterm(trees);
    if (trace)
        printf(" Finding unused non_terminals.\n"), fflush(stdout);

    find_unused_productions(trees);
    if (trace)
        printf(" Finding left recursion.\n"), fflush(stdout);

    find_left_recursion(trees);
    if (trace)
        printf(" Finding maximum K.\n"), fflush(stdout);

    max_K = find_maximum_K(trees);
    if (max_K_input != 0 && max_K > max_K_input)
        max_K = max_K_input;
    for (i = 1; i <= max_K; i++)
    {
        if (trace)
            printf(" Finding first sets for k = %d.\n", i),
                fflush(stdout);

        find_first_sets(trees, i);
        if (trace)
            printf(" Finding follow sets for k = %d.\n", i),
                fflush(stdout);

        find_follow_sets(trees, i);
    }
}

```

```

if (trace)
    printf(" Adjusting first sets for k = %d.\n",l),
        fflush(stdout);
    print_prod(trees);

    find_adjusted_first_sets(trees,l);
    if (trace)
        printf(" Place K into each node for k = %d.\n",l),
            fflush(stdout);
        print_prod(trees);

    NOT_LL = find_K_for_nodes(trees,l);
    if (!NOT_LL)
        break;
    remove_ff_sets(trees);
};
if (NOT_LL)
{
    if ( max_K_input == 0 || max_K < max_K_input )
        yerror("Grammar not LL(k).");
    else
        if (max_K_input != 0)
            yerror("Grammar not LL(k) for any K up to input K.\n");
};
debug("end  decorate()",l);
}

```

```

/* include "struct.h"
/*****

/*
NAME : get_ff_node
PURPOSE : create a first_follow node and initialize it
CALLED BY : copy_ff_set, find_first_sets, calc_first_set, find_follow_sets,
            concat_truncate
CALLS : malloc, sizeof
ARGUMENTS PASSED IN : none
VALUE RETURNED : pointer to created first_follow node
LOCAL VARIABLES :
            t - pointer to the created node
*/

struct first_follow *
get_ff_node()
{
    struct first_follow *t;

    t = (struct first_follow *)malloc(sizeof(struct first_follow));
    t->first_token = NULL;
    t->next_set = NULL;
    return(t);
}
/*****

/*
NAME : get_token_node
PURPOSE : to create a token node
CALLED BY : copy_ff_set, find_first_sets, calc_first_set, find_follow_sets,
            concat_truncate
CALLS : malloc, sizeof
ARGUMENTS PASSED IN : none
VALUE RETURNED : pointer to created token node
LOCAL VARIABLES :
            t - pointer to created node
*/

struct token *
get_token_node()
{
    struct token *t;

    t = (struct token *)malloc(sizeof(struct token));
    t->term_node = NULL;
    t->next_token = NULL;
    t->ref_number = 0;
    return(t);
}
/*****

/*
NAME : copy_ff_set
PURPOSE :
CALLED BY : set_union, calc_first_set, calc_follow_set, first, concat_truncate

```

```

CALLS : get_ff_node, get_token_node
ARGUMENTS PASSED IN : pointer to the first_follow set to be copied
VALUE RETURNED : pointer to the new first_follow set
LOCAL VARIABLES :
    t1,t2,t3 - temporary pointers to tokens
    ret_set - pointer to beginning of set, value to be returned
    f,g - temporary pointers to first sets
*/

struct first_follow *
copy_ff_set(set)
struct first_follow *set;
{
    struct token *t1,*t2,*t3;
    struct first_follow *ret_set,*f,*g;
    for (ret_set = f = g = NULL; set != NULL; set = set->next_set,f = g)
    {
        if (ret_set == NULL)
            ret_set = g = get_ff_node();
        else
            f->next_set = g = get_ff_node();
        for (t1 = t2 = NULL,t3 = set->first_token;
            t3 != NULL;
            t3 = t3->next_token,t1 = t2)
        {
            if (t1 == NULL)
                g->first_token = t2 = get_token_node();
            else
                t1->next_token = t2 = get_token_node();
            t2->term_node = t3->term_node;
            t2->ref_number = t3->ref_number;
        }
        return(ret_set);
    }
}

/*****

NAME : compare_ff_sets
PURPOSE : to compare two first sets, and determine whether the second set
is a subset of the first set
CALLED BY : calc_first_set, calc_follow_set
CALLS : set_equal
ARGUMENTS PASSED IN : pointer to two first/follow sets
VALUE RETURNED : boolean TRUE or FALSE
LOCAL VARIABLES :
    a,b - pointer to a first set
*/

int
compare_ff_sets(set1,set2)
struct first_follow *set1,*set2;
{
    struct first_follow *a,*b;
    if (set1 == NULL && set2 == NULL)
        return(TRUE);

```

```

    if (set1 == NULL || set2 == NULL)
        return(FALSE);
    for (a = set2; a != NULL; a = a->next_set)
    {
        for (b = set1; b != NULL && !set_equal(a,b); b = b->next_set);
        if (b == NULL)
            return(FALSE);
    };
    return(TRUE);
}

/*****
NAME : set_equal
PURPOSE : to compare one specific first/follow item and determine equality
CALLED BY : compare_ff_sets, set_union, adjust_first_sets, calc_first_set,
            replace_non_term_follow_set, adjust_first, first
CALLS : none
ARGUMENTS PASSED IN : pointer to two first set items
VALUE RETURNED : boolean
LOCAL VARIABLES :
*/
a,b - temporary variables to go through sets

int
set_equal(set1,set2)
struct first_follow *set1,*set2;
{
    struct token *a,*b;
    if (set1 == NULL && set2 == NULL)
        return(TRUE);
    if (set1 == NULL || set2 == NULL)
        return(FALSE);
    for (a = set1->first_token,b = set2->first_token;
         a != NULL && b != NULL;
         a = a->next_token,b = b->next_token)
    {
        if (a->term_node->term_num != b->term_node->term_num
            || a->ref_number != b->ref_number)
            break;
    };
    if (a == NULL && b == NULL)
        return(TRUE);
    else
        return(FALSE);
}

/*****
NAME : release_ff_set
PURPOSE : to return a first_follow set to available memory
CALLED BY : set_union, adjust_first_sets, calc_first_set, calc_follow_set,
            replace_non_term_follow_set, adjust_first, first, sub_remove
CALLS : free

```

```

ARGUMENTS PASSED IN : pointer to the set
VALUE RETURNED : none
LOCAL VARIABLES :
    f - temporary pointer to first_follow item
    t1,t2 - temporary pointer to token

*/

release_ff_set(set)
{
    struct first_follow *set;

    struct first_follow *f;
    struct token *t1,*t2;

    while (set != NULL)
    {
        for (t1 = set->first_token; t1 != NULL;
             t2 = t1->next_token;
             free(t1);
             t1 = t2;
             );
        f = set->next_set;
        free(set);
        set = f;
    }
}

/*****
*/
NAME : set_union
PURPOSE : to accept two first sets and merge them
CALLED BY : calc_first_set, calc_follow_set
CALLS : copy_ff_set, set_equal, release_ff_set
ARGUMENTS PASSED IN : two first/follow sets
VALUE RETURNED : pointer to new first/follow set
LOCAL VARIABLES :
    f1,f2,f,g,t - temporary pointers to first set
*/

struct first_follow *
set_union(set1,set2)
{
    struct first_follow *set1,*set2;
    {
        struct first_follow *h1,*h2,*f,*g,*t;
        debug("begin set_union",1);

        if (set1 == NULL)
            return(copy_ff_set(set2));
        if (set2 == NULL)
            return(copy_ff_set(set1));
        h1 = copy_ff_set(set1);
        h2 = copy_ff_set(set2);

        /* find the end of the first set, and attach
           for (f = h1; f->next_set != NULL; f = f->next_set)
           f->next_set = h2;

        /* remove duplicates */

```



```

for (f = hl; f != NULL; f = f->next_set)
{
    for (g = f->next_set; g != NULL; )
    {
        if (set_equal(f, g->next_set))
        {
            t = g->next_set;
            g->next_set = t->next_set;
            t->next_set = NULL;
            release_ff_set(t);
        }
        else
        {
            g = g->next_set;
        }
    };
    debug("end    set_union()", 1);
    return(hl);
}

```

\*\*\*\*\*

```

/*
NAME : set_length
PURPOSE : calculate the length of a particular first set item
CALLED BY : adjust_first_sets, calc_first_set, first
CALLS : none
ARGUMENTS PASSED IN : pointer to first set item
VALUE RETURNED : integer length
LOCAL VARIABLES :
    i - loop counter
    t - loop through tokens
*/

```

```

int set_length(set)
struct first_follow *set;
{
    int i;
    struct token *t;

    if (set == NULL) return(0);
    for (i = 0, t = set->first_token;
         t->next_token != NULL;
         i++, t = t->next_token);
    if (set->first_token->term_node->term_num == -1)
        return(0);
    else
        return(i+1);
}

```

```

# define FDEF_HERE
# include "struct.h"
/*****
/*
NAME : open_files
PURPOSE : to open all temporary and permanent files
CALLED BY : main
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
open_files()
{
    LL_c
    output
    tmp_lex_action
    tmp_prod_action
    var_sub_code
    token_def_code
    tmp_parser
    tmp_trans_table
    = fopen(ll_c,WRITE);
    = fopen(output,WRITE);
    = fopen(tmp_lex_action,WRITE);
    = fopen(tmp_prod_action,WRITE);
    = fopen(tmpvarsub,WRITE);
    = fopen(tmptokendef,WRITE);
    = fopen(tmpparser,WRITE);
    = fopen(tmptranstable,WRITE);
}

/*****
/*
NAME : close_files
PURPOSE : to close all temporary files
CALLED BY : finish_write, yerror
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
close_files()
{
    fflush(stdout);
    fclose(output);
    fclose(tmp_lex_action);
    fclose(tmp_prod_action);
    fclose(var_sub_code);
    fclose(token_def_code);
    fclose(tmp_parser);
    fclose(tmp_trans_table);
}

/*****
/*
NAME : delete_temp_files
PURPOSE : to remove all closed temporary files from the file system
CALLED BY : main, yerror
CALLS : none

```

ARGUMENTS PASSED IN : boolean to designate whether a lexical analyzer was  
 created or not  
 VALUE RETURNED : none  
 LOCAL VARIABLES : none  
 \*/

```

delete_temp_files(lex_made)
int lex_made;
{
    unlink(tmplexaction);
    unlink(tmpproduction);
    unlink(tmpvarsub);
    if (lex_made)
        unlink(tmptokendef);
    unlink(tmpparser);
    unlink(tmptranstable);
}
  
```

```

* include "struct.h"

static k;

/*****

/*
NAME : find_adjusted_first_sets
PURPOSE : to control adjustment of first sets
CALLED BY : decorate
CALLS : adjust_first_sets
ARGUMENTS PASSED IN :
    pointer to the production trees
    maximum_k
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

find_adjusted_first_sets(trees,max_k)
struct production_tree *trees;
int max_k;
{
    debug("begin find_adjusted_first_sets()",1);
    k = max_k;
    for (; trees != NULL; trees = trees->next_tree)
    {
        if (trace)
            printf("\tbegin %s\n",trees->nterm_name),fflush(stdout);
        adjust_first_sets(trees->tree_head);
    };
    debug("end find_adjusted_first_sets()",1);
}

/*****

/*
NAME : adjust_first_sets
PURPOSE :
CALLED BY : find_adjusted_first_sets
CALLS : set_length, concat_truncate, release_ff_set,
    set_equal, adjust_first_sets
ARGUMENTS PASSED IN : pointer to a subtree
VALUE RETURNED : none
LOCAL VARIABLES :
    before - pointer to an ff node before the current one in a loop
    current - pointer to an ff node
    new - pointer to a new ff node
    t1,t2 - temporary pointers to ff nodes
*/

static
adjust_first_sets(tree)
struct tree_node *tree;
{
    struct first_follow *before,*current,*new,*t1,*t2;
    struct first_follow *concat_truncate();
    debug("begin adjust_first_sets()",1);

    new = NULL;

```

```

for (before = NULL, current = tree->first_set; current != NULL;)
{
    if (set_length(current) < k)
    {
        /* concatenate each follow set onto this first set item */
        for (t1 = tree->follow_set; t1 != NULL; t1 = t1->next_set)
        {
            t2 = concat_truncate(current, t1, k);
            t2->next_set = new;
            new = t2;
        };
        t1 = current;
        if (before == NULL)
        {
            tree->first_set = current = current->next_set;
        }
        else
        {
            before->next_set = current = current->next_set;
            t1->next_set = NULL;
            release_ff_set(t1);
        }
    }
    else
    {
        before = current;
        current = current->next_set;
    }
};

if (before == NULL)
{
    tree->first_set = new;
}
else
{
    before->next_set = new;
}

/* remove duplicates */
for (before = tree->first_set;
     before != NULL;
     before = before->next_set)
{
    for (current = before; current->next_set != NULL;)
    {
        if (set_equal(before, current->next_set))
        {
            t1 = current->next_set;
            current->next_set = t1->next_set;
            t1->next_set = NULL;
            release_ff_set(t1);
        }
        else
        {
            current = current->next_set;
        }
    }
};

debug("In adjust_first_sets(after second for)", 1);
switch(tree->node_type)
{
    case empty :
    case terminal :
    case non_terminal :
        break;
    case concat :
    case alternation :
        adjust_first_sets(tree->left, left_child);
        adjust_first_sets(tree->right, right_child);
}

```

```

        break;
    case option :
    case plus :
    case closure :
        adjust_first_sets(tree->left.child);
        break;
    };
    debug("end  adjust_first_sets()",1);
}

```

```

# include "struct.h"

static int FLAG;
static struct first_follow *EMPTY_FIRST_SET;
static int k;

/*****
/*
NAME : find_first_sets
PURPOSE : calculate the first sets for all production trees
CALLED BY : decorate
CALLS : get_ff_node, get_token_node, malloc, calc_first_set
ARGUMENTS PASSED IN :
    pointer to production trees
    maximum K
VALUE RETURNED : none
LOCAL VARIABLES :
    t - pointer to a production tree in loop
    i - integer to count how many iterations in the loop
*/
find_first_sets(prods,max_k)
struct production_tree *prods;
int max_k;
{
    struct production_tree *t;
    struct first_follow *get_ff_node();
    struct token *get_token_node();
    int i;
    debug("begin find_first_sets()",1);

    EMPTY_FIRST_SET = get_ff_node();
    EMPTY_FIRST_SET->first_token = get_token_node();
    EMPTY_FIRST_SET->first_token->term_node =
        (struct term_list *)malloc(sizeof(struct term_list));
    EMPTY_FIRST_SET->first_token->term_node->term_num = -1;
    for (t=prods;t!= NULL; t=t->next_tree)
        t->ptree_mark=0;
    prods->ptree_mark = 1;
    k = max_k;
    i = 1; /* count the iterations */
    FLAG = TRUE;
    while (FLAG)
    {
        if (trace)
            printf("    begin iteration %d.\n",i),fflush(stdout);
        debug("in    find_first_sets begin iteration ",0);
        debug_num(1);
        debug(" ",1);
        FLAG = FALSE;
        if (trace)
            printf("\tbegins %s\n",prods->nterm_name),fflush(stdout);
        calc_first_set(prods->tree_head,prods->nterm_name);
        if (trace)
            printf("\tend    %s\n",prods->nterm_name),fflush(stdout);
        i++;
    };
    prods->ptree_mark = 0;
}

```

```

        debug("end   find_first_sets()",1);
    }

    /*****
    NAME : calc_first_set
    PURPOSE : to calculate the first set for a particular tree node
    CALLED BY : find_first_sets
    CALLS : copy_ff_set, get_ff_node, get_token_node, calc_first_set, first,
            compare_ff_sets, release_ff_set, set_union, yerror, set_length,
            first, set_equal
    ARGUMENTS PASSED IN :
            pointer to subtree
            character string current production name
    VALUE RETURNED : boolean, was the node's first set changed or not
    LOCAL VARIABLES :
            change - integer to see whether the children of a node changed
            ff - temporary pointer to a first set
    */

    static int
    calc_first_set(tree,prod_name)
    struct tree_node *tree;
    char *prod_name;
    {
        struct first_follow *ff;
        struct first_follow *copy_ff_set(),*get_ff_node(),*first(),*set_union();
        struct token *get_token_node();
        int change;
        debug("begin calc_first_set","",0);
        change = FALSE;

        switch(tree->node_type)
        {
            case empty:
                debug("empty",1);
                if (tree->first_set == NULL)
                {
                    tree->first_set = copy_ff_set(EMPTY_FIRST_SET);
                    change = FLAG = TRUE;
                };
                debug("   empty",1);
                break;

            case terminal:
                debug("terminal",1);
                if (tree->first_set == NULL)
                {
                    ((tree->first_set=get_ff_node())->first_token=get_token_node())
                    ->term_node = tree->left.term_ptr;
                    change = FLAG = TRUE;
                };
                debug("   terminal",1);
                break;

            case concat:
                debug("concat",1);
                change = calc_first_set(tree->left.left_child,prod_name)

```



```

        | calc_first_set(tree->right.right_child,prod_name);
        debug("in concat",1);
        if (!change)
            break;
        ff = first(tree->left.left_child->first_set,
                    tree->right.right_child->first_set,k);
        debug("in concat",1);
        if (!compare_ff_sets(tree->first_set,ff))
        {
            release_ff_set(tree->first_set);
            tree->first_set = ff;
            change = FLAG = TRUE;
        }
        else
        {
            release_ff_set(ff);
            change = FALSE;
        };
        debug("concat",1);
        break;
    }

case alternation:
    debug("alternation",1);
    change = calc_first_set(tree->left.left_child,prod_name)
        | calc_first_set(tree->right.right_child,prod_name);
    debug("in alternation",1);
    if (!change)
        break;
    ff = set_union(tree->left.left_child->first_set,
                    tree->right.right_child->first_set,k);
    debug("in alternation",1);
    if (!compare_ff_sets(tree->first_set,ff))
    {
        release_ff_set(tree->first_set);
        tree->first_set = ff;
        change = FLAG = TRUE;
    }
    else
    {
        release_ff_set(ff);
        change = FALSE;
    };
    debug("alternation",1);
    break;

case option:
    debug("option",1);
    change = calc_first_set(tree->left.child,prod_name);
    debug("in option",1);
    if (!change)
        break;
    ff = set_union(tree->left.child->first_set,EMPTY_FIRST_SET);
    debug("in option",1);
    if (!compare_ff_sets(tree->first_set,ff))
    {
        release_ff_set(tree->first_set);
        tree->first_set = ff;
        change = FLAG = TRUE;
    }

```

```

else {
    release_ff_set(ff);
    change = FALSE;
};
debug(" option",1);
break;

case closure:
case plus :
    debug("plus/closure",1);
    change = calc_first_set(tree->left.child,prod_name);
    if (!change)
        break;
    for (ff = tree->left.child->first_set;
         ff != NULL;
         ff = ff->next_set)
    {
        if (ff->first_token->term_node->term_num == -1)
        {
            fprintf(output,"Closure within %s contains empty.\n",
                    prod_name);
            perror ("Closure contains empty");
        };
    };

    int FLAG2;
    struct first_follow *f1,*f2,*t;
    f1 = NULL;
    f2 = ff = copy_ff_set(tree->left.child->first_set);
    debug("in closure",1);
    /* concatenate the first set of the child onto itself,
       all the while saving the result, until one of the results
       has all of its elements with length = k */
    while (1)
    {
        FLAG2 = TRUE;
        for (t = f2; t != NULL && FLAG2; t = t->next_set)
            if (set_length(t) < k)
                FLAG2 = FALSE;
        if (FLAG2)
            break;
        f2 = first(f2,ff,k);
        if (f1 == NULL)
            f1 = f2;
        else
        {
            for (t=f1; t->next_set != NULL; t=t->next_set);
            t->next_set = f2;
        };
        for (t=ff; t != NULL && t->next_set != NULL; t=t->next_set);
        if (tree->node_type == closure)
        {
            /* add the empty string to the first set */
            if (ff == NULL)
            {
                t = copy_ff_set(EMPTY_FIRST_SET);
            }
        }
    }
}

```

```

else {
    t->next_set = f1;
    t = copy_ff_set(EMPTY_FIRST_SET);
};
t->next_set = ff;
ff = t;
};

/* remove duplicates */
for (f1=ff; f1 != NULL; f1=f1->next_set)
{
    for (f2=f1; f2->next_set != NULL; )
    {
        if (set_equal(f1,f2->next_set))
        {
            t=f2->next_set;
            f2->next_set = t->next_set;
            t->next_set = NULL;
            release_ff_set(t);
        }
        else
            f2=f2->next_set;
    }
};

debug("in closure",1);
if (!compare_ff_sets(tree->first_set,ff))
{
    release_ff_set(tree->first_set);
    tree->first_set = ff;
    change = FLAG = TRUE;
}
else {
    release_ff_set(ff);
    change = FALSE;
};
debug(" closure",1);
break;

case non_terminal:
    debug("non_terminal",1);
    if (tree->left.non_term->ptree_mark == 0)
    {
        char *t;
        tree->left.non_term->ptree_mark = 1;
        t = prod_name;
        if (trace)
            printf("\tbegIn %s\n", tree->left.non_term->nterm_name),
                fflush(stdout);
        calc_first_set(tree->left.non_term->tree_head,
                        tree->left.non_term->nterm_name);
        if (trace)
            printf("\tend %s\n", tree->left.non_term->nterm_name),
                fflush(stdout);
        prod_name = t;
        tree->left.non_term->ptree_mark = 0;
    };
    debug("in non_terminal",1);

```

```

if (lcompare_ff_sets(tree->first_set,
{
tree->left.non_term->tree_head->first_set))
{
release_ff_set(tree->first_set);
tree->first_set =
copy_ff_set(tree->left.non_term->tree_head->first_set);
change = FLAG = TRUE;
}
else
{
change = FALSE;
};
debug(" non_terminal",1);
break;
};
print_ff_set(tree->first_set,2);
debug("end calc_first_set()",1);
return(change);
}

```

```

# include "struct.h"

static struct term_list *ZERO;
static int FLAG;
static int k;

/***** */
/*
NAME : find_follow_sets
PURPOSE : to find follow sets for all production trees
CALLED BY : decorate
CALLS : malloc, get_ff_node, get_token_node, calc_follow_set
ARGUMENTS PASSED IN :
    production trees
    maximum k
VALUE RETURNED : none
LOCAL VARIABLES :
    t
    tok, tok2 - to create initial follow set of all EOF tokens
    i - to count number of iterations
*/

find_follow_sets(prods, max_k)
struct production_tree *prods;
int max_k;
{
    struct production_tree *t;
    struct first_follow *get_ff_node();
    struct token *tok, *tok2, *get_token_node();
    int i;
    debug("begin find_follow_sets()", 1);
    k = max_k;
    tok = NULL;

    /* create the list of EOFs for the start symbol's follow set */
    ZERO = (struct term_list *) malloc(sizeof(struct term_list));
    ZERO->term_name = "END_OF_FILE";
    ZERO->term_num = 0;
    for (i=0; i!= k; i++)
    {
        tok2 = get_token_node();
        tok2->next_token = tok;
        tok2->term_node = ZERO;
        tok = tok2;
    };
    tok->ref_number = 1;
    prods->tree_head->follow_set = get_ff_node();
    prods->tree_head->follow_set->first_token = tok;
    prods->ptree_mark = 1;
    i = 1; /* count the iterations */
    FLAG = TRUE;
    while (FLAG)
    {
        if (trace)
            printf("    begin iteration %d.\n", i);
        debug("in    find_follow_sets begin iteration ", i);
        debug_num(1);
        debug("++", 1);
    }
}

```

```

FLAG = FALSE;
if (trace)
    printf("\tbegin %s\n", prods->nterm_name), fflush(stdout);
calc_follow_set(prods->tree_head, TRUE);
if (trace)
    printf("\tend   %s\n", prods->nterm_name), fflush(stdout);
i++;
};
prods->ptree_mark = 0;
debug("end find_follow_sets()", 1);
}

/*****
NAME : calc_follow_set
PURPOSE : to calculate the follow set for a tree node
CALLED BY : find_follow_set
CALLS : first, copy_ff_set, set_union, calc_follow_set, compare_ff_sets,
         release_ff_set, replace_non_term_follow_set
ARGUMENTS PASSED IN :
    pointer to sub tree
    boolean to indicate was the parent changed or not
VALUE RETURNED : none
LOCAL VARIABLES :
    ff - temporary pointer to follow set
*/

static
calc_follow_set(tree, change)
struct tree_node *tree;
int change;
{
    struct first_follow *ff;
    struct first_follow *first();
    debug("begin calc_follow_set(", 0);
    switch(tree->node_type)
    {
        case empty:
            debug("empty", 1);
            print_f_set(tree->follow_set, 2);
            break;

        case terminal:
            debug("terminal", 1);
            print_f_set(tree->follow_set, 2);
            break;

        case concat:
            debug("concat", 1);
            print_f_set(tree->follow_set, 2);
            if (!change)
            {
                calc_follow_set(tree->left, left_child, FALSE);
                calc_follow_set(tree->right, right_child, FALSE);
                break;
            }
            ff = first(tree->right_child->first_set,

```

```

        tree->follow_set,k);
    debug("in  concat (compare left side)",1);
    if (!compare_ff_sets(tree->left_child->follow_set,ff))
    {
        release_ff_set(tree->left_child->follow_set);
        tree->left_child->follow_set = ff;
        change = FLAG = TRUE;
    }
    else
    {
        release_ff_set(ff);
        change = FALSE;
    };
    debug("in  concat (compare right side)",1);
    if (!compare_ff_sets(tree->right_child->follow_set,
        tree->follow_set))
    {
        release_ff_set(tree->right_child->follow_set);
        FLAG = TRUE;
        change = TRUE;
        tree->right_child->follow_set =
            copy_ff_set(tree->follow_set);
    };
    debug("in  concat (done comparing)",1);
    calc_follow_set(tree->left_child,change);
    calc_follow_set(tree->right_child,change);
    break;

case alternation:
    debug("alternation",1);
    print_f_set(tree->follow_set,2);
    debug("in  alternation (compare)",1);
    if (!change)
    {
        calc_follow_set(tree->left_child,FALSE);
        calc_follow_set(tree->right_child,FALSE);
        break;
    };
    change = FALSE;
    if (!compare_ff_sets(tree->left_child->follow_set,
        tree->follow_set))
    {
        release_ff_set(tree->left_child->follow_set);
        release_ff_set(tree->right_child->follow_set);
        tree->left_child->follow_set =
            copy_ff_set(tree->follow_set);
        tree->right_child->follow_set =
            copy_ff_set(tree->follow_set);
        change = FLAG = TRUE;
    };
    debug("in  alternation (done compare)",1);
    calc_follow_set(tree->left_child,change);
    calc_follow_set(tree->right_child,change);
    debug("alternation",1);
    break;

case option:
    debug("option",1);
    print_f_set(tree->follow_set,2);

```

```

debug("In option (compare)",1);
if (!change)
{
    calc_follow_set(tree->left.child,FALSE);
    break;
};
change = FALSE;
if (!compare_ff_sets(tree->left.child->follow_set,
                    tree->follow_set))
{
    release_ff_set(tree->left.child->follow_set);
    tree->left.child->follow_set=copy_ff_set(tree->follow_set);
    change = FLAG = TRUE;
};
debug("In option (done compare)",1);
calc_follow_set(tree->left.child,change);
debug(" option",1);
break;

case closure:
case plus :
    debug("plus/closure",1);
    print_f_set(tree->follow_set,2);
    debug("In closure(compare)",1);
    if (!change)
    {
        calc_follow_set(tree->left.child,FALSE);
        break;
    };
    change = FALSE;
    if (!compare_ff_sets(tree->left.child->follow_set,
                        tree->follow_set))
    {
        ff=set_union(tree->left.child->first_set,tree->follow_set);
        release_ff_set(tree->left.child->follow_set);
        tree->left.child->follow_set = ff;
        change = FLAG = TRUE;
    };
    debug("In closure (done compare)",1);
    calc_follow_set(tree->left.child,change);
    debug(" closure",1);
    break;

case non_terminal:
    debug("non_terminal",1);
    print_f_set(tree->follow_set,2);
    ff = copy_ff_set(tree->follow_set);
    {
        struct first_follow *f;
        struct token *t;
        for (f=ff;f!= NULL;f=f->next_set)
        {
            for (t=f->first_token;t!= NULL;t=t->next_token)
            {
                t->ref_number = 0;
                f->first_token->ref_number = tree->right.ref_num;
            };
        };
    };
    debug("In non_terminal (compare)",1);
    if (!compare_ff_sets(

```



```

        tree->left.non_term->tree_head->follow_set,ff))
    {
        replace_non_term_follow_set(
            tree->left.non_term->tree_head,ff,tree->right.ref_num);
        FLAG = TRUE;
    }
    else
        release_ff_set(ff);
    debug("in non_terminal (done compare)",1);
    if (tree->left.non_term->ptree_mark == g)
    {
        tree->left.non_term->ptree_mark = 1;
        if (trace)
            printf("\tbegin %s\n",tree->left.non_term->nterm_name),
                fflush(stdout);
        calc_follow_set(tree->left.non_term->tree_head,TRUE);
        if (trace)
            printf("\tend %s\n",tree->left.non_term->nterm_name),
                fflush(stdout);
        tree->left.non_term->ptree_mark = g;
    };
    debug(" non_terminal",1);
    break;
};
debug("end calc_follow_set()",1);
}

```

```

/*****

```

```

/*
NAME : replace_non_term_follow_set
PURPOSE : to replace the appropriate follow set items for a non-terminal
          given the reference number
CALLED BY : calc_follow_set
CALLS : release_ff_set, set_equal
ARGUMENTS PASSED IN :
    pointer to non-terminal head node
    pointer to new follow subset
    integer reference number
VALUE RETURNED : none
LOCAL VARIABLES :
    f,t,g - temporary pointers to follow set items
*/

```

```

static
replace_non_term_follow_set(head,new,ref)
struct tree_node *head;
struct first_follow *new;
int ref;
{
    struct first_follow *f,*t,*g;
    debug("begin replace_non_term_follow_set()",1);
    /* delete the old follow sets for this ref number */
    for (g = f = head->follow_set; f != NULL; )
    {
        if (f->first_token->ref_number == ref)
        {
            if (f==head->follow_set)

```

```

        g = head->follow_set = g->next_set;
    else
        g->next_set = f->next_set;
        f->next_set = NULL;
        release_ff_set(f);
    }
    else
        g = f;
        f = g->next_set;
    };

    /* add follow set to old follow set */
    for (f = head->follow_set;
         f != NULL && f->next_set != NULL;
         f = f->next_set);
    if (f == NULL)
        head->follow_set = new;
    else
        f->next_set = new;

    /* remove duplicates */
    for (f = head->follow_set; f != NULL; f = f->next_set)
    {
        for (g = f; g->next_set != NULL; )
        {
            if (set_equal(f, g->next_set))
            {
                t = g->next_set;
                g->next_set = t->next_set;
                t->next_set = NULL;
                release_ff_set(t);
            }
            else
                g = g->next_set;
        }
    };
    debug("end   replace_non_term_follow_set()", 1);
}

```

```

# include "struct.h"

static int NOT_LL;
static char *prod_name;
static int cur_K;

/*****

/*
NAME : find_K_for_nodes
PURPOSE : to calculate the value of k for each node in each production tree
CALLED BY : decorate
CALLS : sub_find_K, adjust_first, final_adjust_first
ARGUMENTS PASSED IN :
    pointer to the production trees
    integer value for the current k being processed
VALUE RETURNED : boolean TRUE if not LL(k), FALSE if LL(k)
LOCAL VARIABLES :
    p - pointer to production tree in loop
*/

int
find_K_for_nodes(prods,current_K)
struct production_tree *prods;
int current_K;
{
    struct production_tree *p;
    debug("begin find_K_for_nodes()",1);
    NOT_LL = FALSE;
    cur_K = current_K;

    for (p=prods; p!= NULL; p=p->next_tree)
    {
        prod_name = p->nterm_name;
        if (trace)
            printf("\t%s\n",prod_name),fflush(stdout);
        sub_find_K(p->tree_head);
    };
    debug("in find_K_for_nodes(after finding k for nodes)",1);
    if (NOT_LL)
        return(TRUE);
    if (trace)
        printf(" Adjust first sets\n"),fflush(stdout);
    for (p=prods; p!= NULL; p=p->next_tree)
    {
        debug("in find_K_for_nodes (begin new production)",1);
        if (trace)
            printf("\tbegin %s\n",p->nterm_name),fflush(stdout);
        adjust_first(p->tree_head,0);
        final_adjust_first(p->tree_head);
    };
    debug("end find_K_for_nodes()",1);
    return(FALSE);
}

/*****/
/*

```

```

NAME : sub_find_K
PURPOSE : to find the value of k for a specific node
CALLED BY : find_K_for_nodes
CALLS : sub_find_K, not_equal_spot
ARGUMENTS PASSED IN : pointer to subtree
VALUE RETURNED : none
LOCAL VARIABLES :
    i - remembers where a first set item is not equal to another item
    f,g - loop variables to go through first sets
*/

static
sub_find_K(tree)
{
    struct tree_node *tree;
    int i;
    struct first_follow *f,*g;
    debug("begin sub_find_K(",0);
    switch(tree->node_type)
    {
        case concat:
            debug("concat",1);
            tree->K = 0;
            sub_find_K(tree->left.left_child);
            sub_find_K(tree->right.right_child);
            break;

        case empty:
            break;

        case terminal:
            debug("empty/terminal/non_terminal",1);
            tree->K = 0;
            break;

        case alternation:
            debug("alternation",1);
            tree->K = 0;
            for (f=tree->left.left_child->first_set;
                f != NULL;
                f = f->next_set)
            {
                for (g=tree->right.right_child->first_set;
                    g != NULL;
                    g = g->next_set)
                {
                    if ((i = not_equal_spot(f,g)) == 0)
                    {
                        fprintf(output,"%s is not LL(%d)\n",
                            prod_name,cur_K);
                        printf("%s is not LL(%d)\n",
                            prod_name,cur_K), fflush(stdout);
                        debug(prod_name,0);
                        debug(" is not LL(k)",1);
                        NOT_LL = TRUE;
                    }
                    if (i > tree->K)
                        tree->K = i;
                }
            }
            sub_find_K(tree->left.left_child);
            sub_find_K(tree->right.right_child);
    }
}

```

```

break;
case closure:
case plus:
case option:
    debug("plus/closure/option",1);
    tree->K = 0;
    for (f=tree->left.child->first_set;
        f != NULL;
        f = f->next_set)
    {
        for (g=tree->follow_set;
            g != NULL;
            g = g->next_set)
        {
            if ((l = not_equal_spot(f,g)) == 0)
            {
                fprintf(output,"%s is not LL(%d)\n",
                    prod_name,cur_K);
                printf("%s is not LL(%d)\n",
                    prod_name,cur_K),fflush(stdout);
                NOT_LL = TRUE;
            };
            if (l > tree->K)
                tree->K = l;
        };
        sub_find_K(tree->left.child);
        break;
    }
    debug("end   sub_find_K()",1);
}

/*****
NAME : not_equal_spot
PURPOSE : to calculate the spot that two first set items are not equal
CALLED BY : sub_find_K
CALLS : none
ARGUMENTS PASSED IN : pointers to the two first set items
VALUE RETURNED : integer number of k for these two, or zero if equal
LOCAL VARIABLES :
    t1,t2 - used in loops through tokens in first sets
    i - holds how many tokens have been looked at so far
*/

static int
not_equal_spot(set1,set2)
struct first_follow *set1,*set2;
{
    struct token *t1,*t2;
    int i;
    debug("begin not_equal_spot()",1);
    for (i=1, t1 = set1->first_token, t2 = set2->first_token;
        (t1 != NULL) && (t2 != NULL) &&
            (t1->term_node->term_num == t2->term_node->term_num);
        t1 = t1->next_token, t2 = t2->next_token, i++);
        if (t1 == NULL && t2 == NULL)

```

```

l = g;
debug("end not_equal_spot(",g);
debug_num(1);
debug(")",1);
return(1);
}

```

```

/*****

```

```

*/

```

```

NAME : final_adjust_first
PURPOSE : to cut all first sets down to the absolute minimum size necessary
CALLED BY : find_K_for_nodes
CALLS : adjust_first, final_adjust_first
ARGUMENTS PASSED IN : pointer to sub_tree
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

```

```

static
final_adjust_first(tree)
struct tree_node *tree;
{
    debug("begin final_adjust_first(",0);
    switch(tree->node_type)
    {
        case empty:
        case terminal:
        case non_terminal:
            debug("empty/terminal/non_terminal",1);
            break;
        case concat:
        case alternation:
            debug("concat/alternation",1);
            adjust_first(tree->left.left_child,tree->K);
            adjust_first(tree->right.right_child,tree->K);
            final_adjust_first(tree->left.left_child);
            final_adjust_first(tree->right.right_child);
            break;
        case option:
        case closure:
        case plus:
            debug("plus/option/closure",1);
            adjust_first(tree->left_child,tree->K);
            final_adjust_first(tree->left_child);
            break;
    }
    debug("end final_adjust_first()",1);
}

```

```

/*****

```

```

*/

```

```

NAME : adjust_first
PURPOSE : to remove the follow set and cut down the first set of a particular
node down to size
CALLED BY : final_adjust_first, find_K_for_nodes
CALLS : release_ff_set, set_equal, free
ARGUMENTS PASSED IN :

```

```

pointer to the tree node
integer value of k for the node
VALUE RETURNED : none
LOCAL VARIABLES :
f,g,h - pointers to first set items during removal of duplicate
items after adjustment
t1,t2 - pointers to tokens in loops
*/
static
adjust_first(tree,k)
struct tree_node *tree;
int k;
{
    int i;
    struct first_follow *f,*g,*h;
    struct token *t1,*t2;
    debug("begin adjust_first()",1);
    release_ff_set(tree->follow_set);
    tree->follow_set = NULL;
    if (k==0)
    {
        release_ff_set(tree->first_set);
        tree->first_set = NULL;
    };
    /* for each first item */
    for (f = tree->first_set; f != NULL; f = f->next_set)
    {
        for (j=1, t1 = f->first_token;
            j < k;
            j++, t1 = t1->next_token);
        t2 = t1->next_token;
        t1->next_token = NULL;
        while (t2 != NULL)
        {
            t1 = t2->next_token;
            free(t2);
            t2 = t1;
        };
    };
    for (f = tree->first_set; f != NULL; f = f->next_set)
    for (g=f; g->next_set != NULL; )
    if (set_equal(f,g->next_set))
    {
        h = g->next_set;
        g->next_set = h->next_set;
        h->next_set = NULL;
        release_ff_set(h);
    }
    else
    {
        g = g->next_set;
    }
    debug("end adjust_first()",1);
}

```

```

# include "struct.h"

static int max_K = 1;

/*****

/*
NAME : find_maximum_K
PURPOSE : to find the maximum k for the trees recursively
CALLED BY : decorate
CALLS : sub_find_k
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : integer value of maximum k
LOCAL VARIABLES :
s - pointer to tree head of start symbol
max_K - variable to hold temporary value of maximum K. Global
to all routines in this file.
*/

int
find_maximum_K(trees)
struct production_tree *trees;
{
    struct tree_node *s;
    debug("begin find_maximum_k()",1);

    s = trees->tree_head;
    trees->ptree_mark = 1;
    for (trees = trees->next_tree; trees != NULL; trees=trees->next_tree)
        trees->ptree_mark = 0;
    sub_find_k(s);
    if (trace)
        printf("\tMaximum k = %d\n",max_K),fflush(stdout);
    debug("end find_maximum_k()",0);
    debug_num(max_K);
    debug(")",1);
    return(max_K);
}

/*****

/*
NAME : sub_find_k
PURPOSE : to calculate the maximum k at a tree node
CALLED BY : find_maximum_K
CALLS : sub_find_k
ARGUMENTS PASSED IN : pointer to tree node
VALUE RETURNED : integer value of maximum k
LOCAL VARIABLES :
i,j - values of left and right child, respectively
*/

static int
sub_find_k(tree)
struct tree_node *tree;
{
    /*return i*/
    int i,j;
    debug("begin sub_find_k()",1);

```



THIS PAGE LEFT INTENTIONALLY BLANK

```

switch(tree->node_type)
{
    case empty:
        i = g;
        break;
    case terminal:
        i = l;
        break;
    case concat:
        i = sub_find_k(tree->left, left_child);
        j = sub_find_k(tree->right, right_child);
        if (i >= g)
            if (j < g)
                i = j-1;
            else
                i = j+1;
        else
            break;
    case option:
        break;
    case closure:
        break;
    case plus:
        i = sub_find_k(tree->left, child);
        break;
    case alternation:
        i = sub_find_k(tree->left, left_child);
        j = sub_find_k(tree->right, right_child);
        if (i < g)
            if (j < g)
                {
                    if (i > j)
                        i = j;
                }
            else
                {
                    if ((i=i*-1) < j)
                        i = j;
                }
        else
            if (j < g)
                {
                    if (i < (j=j*-1))
                        i = j;
                }
            else
                {
                    if (i < j)
                        i = j;
                }
        break;
    case non_terminal:
        if (tree->left.non_term->ptree_mark == 1)
            i = -1;
        else
            {
                tree->left.non_term->ptree_mark = 1;
                i = sub_find_k(tree->left.non_term->tree_head);
                tree->left.non_term->ptree_mark = g;
            }
        break;
}

```

```
);  
if (-i > max_k) max_k = -i;  
if ( i > max_k) max_k = i;  
debug("end sub_find_k(",0);  
debug_num(i);  
debug("-"),1);  
return(i);  
}
```

```

* include "struct.h"
/*****
/*
NAME : find_left_recursion
PURPOSE : to find any left recursion and report it as an error
CALLED BY : decorate
CALLS : sub_find_recursion
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : none
LOCAL VARIABLES :
t - loop variable to go through production trees
*/

find_left_recursion(prod)
struct production_tree *prod;
{
    struct production_tree *t;
    debug("begin find_left_recursion()",1);
    for (t=prod; t!= NULL; t=t->next_tree)
        t->ptree_mark = 0;
    for (t=prod; t!= NULL; t=t->next_tree)
        /* if the production has not been looked at already */
        if (t->ptree_mark != 1)
            sub_find_recursion(t->tree_head);
    debug("end find_left_recursion()",1);
}
/*****
/*
NAME : sub_find_recursion
PURPOSE : to recursively go through trees and if recursion exists, report it
CALLED BY : find_left_recursion
CALLS : sub_find_recursion, yerror
ARGUMENTS PASSED IN : pointer to tree node
VALUE RETURNED : 0 if it is possible that no terminal might be visited when
this path is taken, 1 otherwise
LOCAL VARIABLES :
i - temporary variable to hold the return value
*/

static int
sub_find_recursion(tree)
struct tree_node *tree;
{
    /* returns 0 if possible for no terminals to be visited, 1 o.w.*/
    int i;
    debug("begin sub_find_recursion()",1);
    switch(tree->node_type)
    {
        case closure:
        case plus:
        case option:
            sub_find_recursion(tree->left.child);
            i = 0;
            break;

```

```

case alternation:
    i = sub_find_recursion(tree->left_child)
    & sub_find_recursion(tree->right_child);
    break;
case concat:
    if (sub_find_recursion(tree->left_child) == 0)
        i = sub_find_recursion(tree->right_child);
    else
        i = 1;
    break;
case terminal:
    i = 1;
    break;
case empty:
    i = 0;
    break;
case non_terminal:
    if (tree->left.non_term->ptree_mark == 1)
        i = sub_find_recursion(tree->left.non_term->tree_head);
    if (tree->left.non_term->ptree_mark == -1)
    {
        fprintf(output, "Left recursion on %s\n",
            tree->left.non_term->nterm_name);
        yyerror("Left recursion found");
    };
    if (tree->left.non_term->ptree_mark == 0)
    {
        if (tree->left.non_term->ptree_mark == 0)
        {
            tree->left.non_term->ptree_mark = -1;
            i = sub_find_recursion(tree->left.non_term->tree_head);
            tree->left.non_term->ptree_mark = 1;
        };
        break;
    };
    debug("end sub_find_recursion(", 0);
    debug_num(i);
    debug(")=", 1);
    return(i);
}

```

```

# include "struct.h"
/*****
/*
NAME : find_and_number_references
PURPOSE : to uniquely number all terminal references
CALLED BY : decorate
CALLS : sub_number_refs
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : none
LOCAL VARIABLES :
t - temporary pointer to production trees
*/
find_and_number_references(trees)
struct production_tree *trees;
{
    struct production_tree *t;
    debug("begin find_and_number_references()",1);
    for (t=trees; t != NULL; t=t->next_tree)
        t->ptree_mark = 1;
    trees->ptree_mark = 2;
    for (t=trees; t!= NULL; t=t->next_tree)
        sub_number_refs(t->tree_head);
    debug("end find_and_number_references()",1);
}
/*****
/*
NAME : sub_number_refs
PURPOSE : to recursively peruse the trees, and mark the non_terminal nodes
CALLED BY : find_and_number_references
CALLS : sub_number_refs
ARGUMENTS PASSED IN : pointer to a subtree
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
static
sub_number_refs(tree)
struct tree_node *tree;
{
    debug("begin sub_number_refs()",1);
    switch(tree->node_type)
    {
        case closure:
        case plus:
        case option:
            sub_number_refs(tree->left.child);
            break;
        case alternation:
        case concat:
            sub_number_refs(tree->left.left_child);
            sub_number_refs(tree->right.right_child);
    }
}
/*****/

```

```

        break;
    case empty:
    case terminal:
        break;
    case non_terminal:
        tree->right.ref_num =
            tree->left.non_term->ptree_mark++;
        break;
    };
    sub_number_refs(1);
    return;
}

```

```

* include "struct.h"

/*
NAME : find_undefined_nonterm
PURPOSE : to find all undefined non_terminals
CALLED BY : decorate
CALLS : none
ARGUMENTS PASSED IN : pointer to production tree
VALUE RETURNED : none
LOCAL VARIABLES :
    FLAG - boolean to remember if one was undefined
*/

find_undefined_nonterm(prod)
struct production_tree *prod;
{
    int FLAG;
    debug("begin find_undefined_nonterm()",1);

    FLAG = FALSE;
    for (; prod != NULL; prod = prod->next_tree)
    {
        if (prod->tree_head == NULL)
        {
            FLAG = TRUE;
            fprintf(output,"%s is undefined\n",prod->nterm_name);
        }
    }

    if (FLAG)
        yyerror("Undefined non_terminals exist");
    debug("end find_undefined_nonterm",1);
}

```



```

# include "struct.h"
/*****
/*
NAME : find_unused_productions
PURPOSE : to find any productions that are never used
CALLED BY : decorate
CALLS : mark_unused_productions
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : none
LOCAL VARIABLES :
t - loop pointer to a production tree
FLAG - remembers whether any unmarked production became marked in a
loop iteration
*/
find_unused_productions(trees)
struct production_tree *trees;
{
    struct production_tree *t;
    int FLAG;
    debug("begin find_unused_productions()",1);
    for (t=trees;t!= NULL; t=t->next_tree)
        t->ptree_mark = 0;
    trees->ptree_mark = 1;
    FLAG = TRUE;
    while (FLAG)
    {
        for (t=trees,FLAG=FALSE;t!= NULL;t=t->next_tree)
            FLAG = mark_unused_productions(t->tree_head) || FLAG;
    };

    for (t=trees,FLAG=FALSE;t!= NULL; t=t->next_tree)
        if (t->ptree_mark == FALSE)
        {
            FLAG = TRUE;
            fprintf(output,"%s is never used\n",t->nterm_name);
        };
        if (FLAG)
            yyerror("Unused productions exist");
        debug("end find_unused_productions()",1);
    }
}
/*****
/*
NAME : mark_unused_productions
PURPOSE : to recursively find non-terminal references and mark the non-terminal
CALLED BY : find_unused_productions
CALLS : mark_unused_productions
ARGUMENTS PASSED IN : pointer to subtree
VALUE RETURNED : boolean, a non-terminal was marked or not
LOCAL VARIABLES :
FLAG - temporary to remember whether one was marked
*/
static int

```

```

mark_used Productions(tree)
struct tree_node *tree;
{
    int FLAG;
    debug("begin mark_used Productions()", 1);
    switch(tree->node_type)
    {
        case empty:
        case terminal:
            FLAG = FALSE;
            break;
        case non_terminal:
            if (tree->left.non_term->ptree_mark == FALSE)
                FLAG = tree->left.non_term->ptree_mark = TRUE;
            else
                FLAG = FALSE;
            break;
        case option:
        case closure:
        case plus:
            FLAG = mark_used Productions(tree->left.child);
            break;
        case alternation:
        case concat:
            FLAG = mark_used Productions(tree->left.left_child)
                | mark_used Productions(tree->right.right_child);
            break;
    };
    debug("end mark_used Productions()", 1);
    return(FLAG);
}

```

```

# include "estruct.h"

# define MAX_FILES 11
static char *files[MAX_FILES];

/*
NAME : finish_write
PURPOSE : to take all current output files and organize them for final output
CALLED BY : main
CALLS : close_file
ARGUMENTS PASSED IN :
VALUE RETURNED : none
LOCAL VARIABLES :
files - array of file names
i - integer, loops through file name array
c - character used to read from and write data
file - temporary to hold fcb for one file
*/

finish_write(lex_made)
int lex_made;
{
    int i;
    char c;
    FILE *file;
    debug("begin finish_write()",1);

    fprintf(tmp_parser, "\n\n");
    fprintf(var_sub_code, "\n\n");
    fprintf(token_def_code, "# define END_OF_FILE\t0\n");
    close_files();

    files[0] = tmpvarsub;
    files[1] = tmptokendef;
    files[2] = code_file_1;
    files[3] = code_file_2;
    files[4] = tmptranstable;
    files[5] = code_file_3;
    files[6] = tmplexaction;
    files[7] = code_file_4;
    files[8] = tmpparser;
    files[9] = tmpproduction;
    files[10] = code_file_5;

    fprintf(LL_c, "\n");
    for (i=0; i<MAX_FILES; i++)
    {
        if (lex_made || (i!=3 && i!=5 && i!=7))
        {
            file = fopen(files[i], "r");
            while ((c = fgetc(file)) != EOF)
                fprintf(LL_c, "%c", c);
            fclose(file);
        }
    };
    fprintf(LL_c, "\n\n");
    debug("end finish_write()", 1);
}

```

```

# include "struct.h"
/*****
*/
NAME : first
PURPOSE : to concatenate two complete first sets together, and cut off at k
CALLED BY : calc_first_set, calc_follow_set
CALLS : set_length, concat_truncate, copy_ff_set, set_equal
ARGUMENTS PASSED IN :
    pointer to beginning set
    pointer to ending set
    value for k
VALUE RETURNED : pointer to new first/follow set
LOCAL VARIABLES :
    ff - holds pointer to new first/follow set
    a,b,c - temporary pointers to first/follow sets
*/

struct first_follow *
first(set1,set2,k)
struct first_follow *set1,*set2;
int k;
{
    struct first_follow *a,*b,*c,*ff;
    struct first_follow *concat_truncate(),*copy_ff_set();
    debug("begin first()",1);

    ff = NULL;
    /* concatenate the second set on the first in all possible combinations */
    for (a=set1; a != NULL; a = a->next_set)
    {
        if (set_length(a) < k)
        {
            for (b = set2; b != NULL; b=b->next_set)
            {
                c = concat_truncate(a,b,k);
                c->next_set = ff;
                ff = c;
            }
        }
        else
        {
            c = copy_ff_set(a);
            c->next_set = ff;
            ff = c;
        }
    };
    /* remove duplicates */
    for (a=ff; a != NULL; a=a->next_set)
    {
        for (b=a; b->next_set != NULL; )
        {
            if (set_equal(a,b->next_set))
            {
                c=b->next_set;
                b->next_set = c->next_set;
                c->next_set = NULL;
                release_ff_set(c);
            }
        }
    }
}

```

```

    )
    else {
        b=b->next_set;
    };
};

if (ff == NULL && set1 == NULL)
    ff = copy_ff_set(set2);
if (ff == NULL && set2 == NULL)
    ff = copy_ff_set(set1);
debug("end first()",1);
return(ff);
}

/*****
NAME : concat_truncate
PURPOSE : accept two first set items, concatenate them, and cut them off at k
CALLED BY : adjust_first_sets, first
CALLS : get_ff_node, get_token_node, copy_ff_set
ARGUMENTS PASSED IN :
    pointer to beginning set item
    pointer to ending set item
    value of k
VALUE RETURNED : pointer to new first set item
LOCAL VARIABLES :
    ret_set - pointer to new first/follow set
    t1,t2,t3 - pointers to tokens
    i - integer remembers how many tokens have been placed on first item
*/

struct first_follow *
concat_truncate(set1,set2,k)
struct first_follow *set1,*set2;
int k;
{
    struct first_follow *ret_set,*get_ff_node();
    struct token *t1,*t2,*t3,*get_token_node();
    int i;
    if (set1 == NULL)
    {
        ret_set = copy_ff_set(set2);
        return(ret_set);
    };
    if (set2 == NULL)
    {
        ret_set = copy_ff_set(set1);
        return(ret_set);
    };

    /* place the nodes of the first string onto the result */
    for (ret_set = get_ff_node(), i=0, t1 = t2 = NULL,t3=set1->first_token;
         t3 != NULL && i < k && t3->term_node->term_num != -1;
         t3 = t3->next_token, i++, t1 = t2)
    {
        if (t1 == NULL)
            ret_set->first_token = t2 = get_token_node();

```

```

else
    t1->next_token = t2 = get_token_node();
    t2->term_node = t3->term_node;
    t2->ref_number = t3->ref_number;
};

/* place the nodes of the second string onto the result */
for (t3 = set2->first_token;
     t3 != NULL && i < k && (t3->term_node->term_num != -1 || i == 0);
     t3 = t3->next_token, i++)
{
    if (t1 == NULL)
        ret_set->first_token = t2 = get_token_node();
    else
        t1->next_token = t2 = get_token_node();
    t2->term_node = t3->term_node;
    t2->ref_number = t3->ref_number;
};
return(ret_set);
}

```

```

# include "struct.h"
/*****

/*
NAME : strcpy
PURPOSE : to take string input and return a pointer to a new copy
CALLED BY : new_prod_node, YACC input
CALLS : malloc
ARGUMENTS PASSED IN : pointer to string
VALUE RETURNED : pointer to string
LOCAL VARIABLES :
    u - pointer to string to return
    t - pointer to run through memory and allow characters to be placed
*/

char *
strcpy (from)
char *from;
{
    char *t,*u;
    u=t=(char *)malloc(strlen(from)+1);
    while (*t++ = *from++);
    return(u);
}

/*****

/*
NAME : strcmp
PURPOSE : compare two strings
CALLED BY : find_prod_node, YACC input, is_id_terminal
CALLS : none
ARGUMENTS PASSED IN : pointers to two strings
VALUE RETURNED : boolean, 1 if =, 0 if not
LOCAL VARIABLES : none
*/

int
strcmp(s,t)
char *s,*t;
{
    for (;*s==*t;s++,t++)
        if (*s=='\0')
            return(TRUE);
    return(FALSE);
}

```

```

X{
/*
NAME : (LEX input)
PURPOSE : to accept tokens from input and pass them to YACC
CALLED BY : (YACC Input)
CALLS : yvwrap(
ARGUMENTS PASSED IN : none
VALUE RETURNED : integer value of a token
SIDE EFFECT : the value of ytext is modified to be the new token
LOCAL VARIABLES :
numbrace - counts the number of braces in an action
*/

# include "yacc.h"
int numbrace = 0;
int trace = 0;
int line_num = 1;

/* global definition of trace */

Xstart varsups prods pcode prodquote lexcode
Xstart lexquote lexbrack lexnorm token defines
XX
<varsups>^\\n
(
yytext[0] = ' ';
line_num++;
return(VAR_SUB_LINE);
)
line_num++;
return(VAR_SUB_LINE);
return(DOLLAR_DOLLAR);
return(DOLLAR_DOLLAR);
Xstart lexquote lexbrack lexnorm token defines
XX
<pcode>^\\n
(
if (numbrace == 1)
{
p_state("prods");
BEGIN prods;
}
numbrace--;
return(RBRACE);
)
<pcode>^{"
(
p_state("prodquote");
BEGIN prodquote;
return(QUOTE);
)
numbrace++;
return(LBRACE);
)
line_num++;
return(RETURN);
)
return(ANY_CHAR);
return(ESCAPED_QUOTE);
(
p_state("pcode");
BEGIN pcode;
return(QUOTE);
)
return(ANY_CHAR);
yyerror("illegal return in quote");
(
p_state("varsups");
line_num++;
BEGIN varsups;
return(PERCENT);
)
return(COLON);
return(SEMICOLON);
return(ALTERNATE);
return(RPAREN);
return(LPAREN);

```



[illegible]

```

        (
            <lexnorm>~^"XX".*~\n
            BEGIN lexcode;
            return(LEX_ACTION_CODE);
            p_state("prods");
            line_num++;
            BEGIN prods;
            return(PERCENT);
            p_state("lexcode");
            line_num++;
            BEGIN lexcode;
            return(LEX_ACTION_CODE);
            return(ANY_CHAR);
            return(OCTAL_NUMBER);
            p_state("lexnorm");
            line_num++;
            BEGIN lexnorm;
            return(PERCENT);
            return(IDENTIFIER);
            line_num++;
            ;
            yyerror("illegal character in token stream");
            {
                line_num++;
                trace = 1;
            }
            p_state("token");
            BEGIN token;
            unput(yytext[0]);
            line_num++;
            {
                p_state("defines");
                BEGIN defines;
                unput(yytext[0]);
            }
        )
    }

    %*
    /*****
    NAME : yywrap
    PURPOSE : to return a 1 when the input file is at EOF
    CALLED BY : (LEX Input)
    CALLS : none
    ARGUMENTS PASSED IN : none
    VALUE RETURNED : 1
    LOCAL VARIABLES : none
    */
    static yywrap()
    {
        return(1);
    }

    /*****
    NAME : p_state
    PURPOSE : prints a change of state in LEX if in debug mode
    CALLED BY : (LEX input)
    CALLS : debug
    ARGUMENTS PASSED IN : new state name
    VALUE RETURNED : none
    LOCAL VARIABLES : none
    */

```

```
static
p_state(state)
char *state;
{
    debug("BEGIN '"',0);
    debug(state,0);
    debug("'"',1);
}
```

```

# include "struct.h"

/*
NAME : make_alternation
PURPOSE : to output the code for an alternation production tree node
CALLED BY : make_alternation, make_concat, make_closure_plus,
            make_proc_non_term, make_option
CALLS : make_alternation, pr_lookahead_if, out_parser, make_closure_plus,
        make_concat, make_option, make_non_terminal, make_terminal,
        make_empty, last_was_closure
ARGUMENTS PASSED IN :
    pointer to subtree
    value of indentation of parser output code
    flag to denote whether the alternation is within a closure or not
VALUE RETURNED : none
LOCAL VARIABLES :
    print_end_set - boolean flag to decide whether or not to print
                    the "LL_end_set" call on the output
*/

make_alternation(tree, indent, within_closure)
struct tree_node *tree;
int indent;
int within_closure;
{
    int print_end_set;
    print_end_set = TRUE;
    debug("begin make_alternation()", 1);
    pr_lookahead_if((tree->left_child->first_set, tree->K, indent);
    out_parser("\n", indent+4);
    switch(tree->left_child->node_type)
    {
        case alternation:
            make_alternation (tree->left_child, indent+4,
                             within_closure);
            /* end-set printed under each alternative's node */
            print_end_set = FALSE;
            break;

        case closure :
        case plus :
            make_closure_plus(tree->left_child, indent+4);
            break;

        case concat :
            make_concat (tree->left_child, indent+4);
            break;

        case option :
            make_option (tree->left_child, indent+4);
            break;

        case non_terminal:
            make_non_terminal(tree->left_child, indent+4);
            break;

        case terminal :
            make_terminal (tree->left_child, indent+4);
            break;

        case empty :
            make_empty (tree->left_child, indent+4);
            break;
    }
}

```

```

if (print_end_set)
{
    out_parser("LL_end_set(set_begin, ", indent+4);
    if (within_closure)
        out_parser("TRUE", indent+4);
    else
        out_parser("FALSE", indent+4);
    out_parser(", first_closure, ", indent+4);
    out_parser(last_was_closure(tree->left_child), indent+4);
    out_parser(");\n", indent+4);
};

out_parser(");\n", indent+4);
out_parser("else\n", indent);
out_parser("{\n", indent+4);
print_end_set = TRUE;

/* now right side */
switch(tree->right_child->node_type)
{
    case alternation:
        make_alternation (tree->right_child, indent+4,
                        within_closure);
        /* end-set printed under each alternative's node */
        print_end_set = FALSE;
        break;

    case closure :
    case plus :
        make_closure_plus(tree->right_child, indent+4);
        break;

    case concat :
        make_concat (tree->right_child, indent+4);
        break;

    case option :
        make_option (tree->right_child, indent+4);
        break;

    case non_terminal:
        make_non_terminal(tree->right_child, indent+4);
        break;

    case terminal :
        make_terminal (tree->right_child, indent+4);
        break;

    case empty :
        make_empty (tree->right_child, indent+4);
        break;
};

if (print_end_set)
{
    out_parser("LL_end_set(set_begin, ", indent+4);
    if (within_closure)
        out_parser("TRUE", indent+4);
    else
        out_parser("FALSE", indent+4);
    out_parser(", first_closure, ", indent+4);
    out_parser(last_was_closure(tree->right_child), indent+4);
    out_parser(");\n", indent+4);
};

out_parser(");\n", indent+4);
debug("end make_alternation()", 1);
}

```

```

struct FSA *ndfsa;
{
    struct FSA *fsa, *to;
    int FLAG;
    debug("begin remove_empty_transitions()",1);

    FLAG = TRUE;
    while (FLAG)
    {
        FLAG = FALSE;
        for (fsa=ndfsa;fsa!=NULL;fsa=fsa->next_state)
        {
            while (fsa->EMPTY_transition != NULL)
            {
                to=fsa->EMPTY_transition->to_state;
                delete_transition(fsa,to,EMPTY);
                merge_into(fsa,to);
                FLAG = TRUE;
            };
        };
        debug("end remove_empty_transitions()",1);
    }
}

/*****
/
NAME : merge_into
PURPOSE : merge transitions from one state into another state
CALLED BY : remove_empty_transitions, make_deterministic
CALLS : add_transition
ARGUMENTS PASSED IN :
    pointer to a state that will have transitions merged into it
    pointer to a state from which the transitions are gotten to merge
VALUE RETURNED : none
LOCAL VARIABLES :
    i - pointer to a transition
    I - integer loop variable
*/

static
merge_into(into_state,from_state)
struct FSA *into_state,*from_state;
{
    struct trans_type *t;
    int i;
    debug("begin merge_into()",1);
    for (i=EMPTY;i<MAX_CHAR_SET;i++)
    {
        switch(i)
        {
            case EMPTY: i=from_state->EMPTY_transition;break;
            default : i=from_state->transition[i];break;
        };
        for (;i!=NULL;i=i->next_trans)
            add_transition(into_state,i->to_state,i);
    };
    if ( (from_state->action_id != 0)
        && ( (into_state->action_id == 0)
    )

```

```

# include "struct.h"

/*
NAME : make_closure_plus
PURPOSE : to output parser code for closure and closure plus nodes
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_non_term, make_option
CALLS : out_parser, pr_lookahead_if, make_alternation, make_closure_plus,
            make_concat, make_option, make_non_terminal, make_terminal,
            make_empty, last_was_closure
ARGUMENTS PASSED IN :
            pointer to production subtree
            current indentation for output code
VALUE RETURNED : none
LOCAL VARIABLES :
            print_end_set - remembers based on whether alternation was
                           below this node whether to print the end set here or not
*/

make_closure_plus(tree, indent)
struct tree_node *tree;
int indent;
{
    int print_end_set;
    print_end_set = TRUE;
    debug("begin make_closure()", 1);

    out_parser("if ( first_closure == NULL )\n", indent);
    out_parser("first_closure = LL_begin_closure();\n", indent+5);
    out_parser("else LL_begin_closure();\n", indent);
    out_parser("while (TRUE)\n", indent);
    out_parser("{\n", indent+4);
    out_parser("int *set_begin, *first_closure;\n", indent+4);
    out_parser("first_closure = NULL;\n", indent+4);
    if (tree->node_type == closure)
    {
        pr_lookahead_if((tree->left.child->first_set, tree->K, indent+4);
        out_parser("{\n", indent+8);
        indent = indent + 8;
    }
    else
        indent = indent + 4;
    out_parser("set_begin = LL_begin_set();\n", indent);

    switch((tree->left.child->node_type)
    {
        case alternation:
            make_alternation (tree->left.child, indent, TRUE);
            /* end-set printed under each alternative's node */
            print_end_set = FALSE;
            break;

        case closure :
        case plus :
            make_closure_plus(tree->left.child, indent);
            break;

        case concat :
            make_concat      (tree->left.child, indent);

```

```

        break;

    case option :
        make_option      (tree->left.child, indent);
        break;

    case non_terminal :
        make_non_terminal(tree->left.child, indent);
        break;

    case terminal :
        make_terminal     (tree->left.child, indent);
        break;

    case empty :
        make_empty        (tree->left.child, indent);
        break;
    };
    if (print_end_set)
    {
        out_parser("LL_end_set(set_begin, TRUE, first_closure, ",
            indent);
        out_parser(last_was_closure(tree->left.child), indent);
        out_parser("):\n", indent);
    };

    if (tree->node_type == closure)
        indent = indent - 8;
    else
    {
        indent = indent - 4;
        pr_lookahead_if(tree->left.child->first_set, tree->K, indent+4);
        out_parser("\n", indent+8);
    };
    out_parser("\n", indent+8);
    out_parser("else\n", indent+4);
    out_parser("{\n", indent+8);
    out_parser("break;\n", indent+8);
    out_parser("):\n", indent+8);
    out_parser("):\n", indent+4);
    debug("end make_closure()", 1);
}

```



```

# include "struct.h"
static char *space = " ";
/*****
*/
/*
NAME : make_reg_expr_code
PURPOSE : to output the code for the FSA
CALLED BY : make_lexical_analyzer
CALLS : one_if
ARGUMENTS PASSED IN : pointer to FSA
VALUE RETURNED : none
LOCAL VARIABLES :
j - temporary pointer to a FSA
i - integer used in loop through characters
*/
make_reg_expr_code(fsa)
struct FSA *fsa;
{
    int i;
    struct FSA *j;
    /* for each state */
    for (;fsa != NULL;fsa=fsa->next_state)
    {
        fprintf(tmp_trans_table,"%16scase %d :\n",space,fsa->state_num);
        fprintf(tmp_trans_table,"%20stack[stackplace].action = %d;\n",
            space,fsa->action_id);
        fprintf(tmp_trans_table,"%20s",space);
        /* for each transition */
        for (i=1;i<MAX_CHAR_SET;i++)
        {
            if (fsa->transition[i] != NULL)
            {
                j=fsa->transition[i]->to_state;
                one_if(j,fsa);
                fprintf(tmp_trans_table,"%24snext_state = %d;\n",space,
                    j->state_num);
                fprintf(tmp_trans_table,"%20selse ",space);
            }
            fprintf(tmp_trans_table,"transition_OK = FALSE;\n%20sbreak;\n",space);
        }
    }
}
/*****
*/
/*
NAME : one_if
PURPOSE : to print the if statement for one state's transitions to one
particular state
CALLED BY : make_reg_expr_code
CALLS : pchar
ARGUMENTS PASSED IN :
pointer to a state that this if statement is to
pointer to the state for which the if statement is being made
VALUE RETURNED : none
LOCAL VARIABLES :
INSIDE - remembers whether a set of transitions in a row all have

```

```

        the same to_state
FIRST_ONE - boolean to remember if the first part of the if has
        been printed yet
        i - loop through the transition array for a state

*/
static
one_if(to_st,state)
struct FSA *to_st, *state;
{
    int i;
    int INSIDE,FIRST_ONE;
    char *pchar();
    INSIDE = FALSE;
    FIRST_ONE = TRUE;

    for (i=1;i<MAX_CHAR_SET;i++)
    {
        if (!INSIDE)
        {
            if (state->transition[i]->to_state == to_st)
            {
                if (FIRST_ONE)
                {
                    fprintf(tmp_trans_table,"if ( ");
                    FIRST_ONE = FALSE;
                }
                else
                {
                    fprintf(tmp_trans_table,"\n%20s|| ",space);
                }
                if (i==MAX_CHAR_SET || to_st!=state->transition[i+1]->to_state)
                {
                    fprintf(tmp_trans_table,"c == %s ",pchar(i));
                    state->transition[i] = NULL;
                }
            }
            else
            {
                /* INSIDE defines that a sequence of transitions
                all have the same to-state */
                INSIDE = TRUE;
                fprintf(tmp_trans_table,"{ c >= %s && ",pchar(i));
                state->transition[i] = NULL;
            }
        }
    }
}
else
{
    if (i==MAX_CHAR_SET || to_st != state->transition[i+1]->to_state)
    {
        fprintf(tmp_trans_table,"c <= %s ) ",pchar(i));
        state->transition[i] = NULL;
        INSIDE = FALSE;
    }
    else
    {
        state->transition[i] = NULL;
    }
}
};

```



```

# include "struct.h"

/*
NAME : make_concat
PURPOSE : to output parser code for a concatenation node
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : out_parser, make_alternation, make_closure_plus, make_concat
ARGUMENTS PASSED IN :
    pointer to subtree
    current indentation of the parser code
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

make_concat(tree, indent)
struct tree_node *tree;
int indent;
{
    debug("begin make_concat()", 1);
    switch(tree->left_child->node_type)
    {
        case alternation:
            out_parser("\n", indent);
            out_parser("int *set_begin, *first_closure; \n",
                indent+4);
            out_parser("first_closure = NULL; \n", indent+4);
            out_parser("set_begin = LL_begin_set(); \n", indent+4);
            make_alternation(tree->left_child, indent+4, FALSE);
            out_parser("); \n", indent);
            break;

        case closure :
        case plus :
            make_closure_plus(tree->left_child, indent);
            break;

        case concat :
            out_parser("\n", indent);
            out_parser("int *set_begin, *first_closure; \n",
                indent+4);
            out_parser("first_closure = NULL; \n", indent+4);
            out_parser("set_begin = LL_begin_set(); \n", indent+4);
            make_concat(tree->left_child, indent+4);
            out_parser("LL_end_set(set_begin, FALSE, first_closure, ",
                indent+4);
            out_parser("last_was_closure(tree->left_child),",
                indent+4);
            out_parser("); \n", indent+4);
            out_parser("); \n", indent);
            break;

        case option :
            make_option(tree->left_child, indent);
            break;

        case non_terminal:
    }
}

```

```

        make_non_terminal(tree->left.left_child,indent);
        break;

    case terminal :
        make_terminal      (tree->left.left_child,indent);
        break;

    case empty :
        make_empty         (tree->left.left_child,indent);
        break;
};

/* now right side */
switch(tree->right.right_child->node_type)
{
    case alternation:
        out_parser("\n",indent);
        out_parser("int *set_begin,*first_closure;\n",
            indent+4);
        out_parser("first_closure = NULL;\n",indent+4);
        out_parser("set_begin = LL_begin_set();\n",indent+4);
        make_alternation(tree->right.right_child,indent+4,
            FALSE);
        out_parser(");\n",indent);
        break;

    case closure :
        break;
    case plus :
        make_closure_plus(tree->right.right_child,indent);
        break;

    case concat :
        make_concat      (tree->right.right_child,indent);
        break;

    case option :
        make_option      (tree->right.right_child,indent);
        break;

    case non_terminal:
        make_non_terminal(tree->right.right_child,indent);
        break;

    case terminal :
        make_terminal     (tree->right.right_child,indent);
        break;

    case empty :
        make_empty        (tree->right.right_child,indent);
        break;
};
debug("end  make_concat()",1);
}

```

```

#include "struct.h"

/*
NAME : make_defs_and_main
PURPOSE : to output the main procedure for the parser
CALLED BY : make_parser
CALLS : out_parser
ARGUMENTS PASSED IN :
    value of k
    pointer to production tree
VALUE RETURNED : none
LOCAL VARIABLES :
    num - temporary character string to hold value of k before output
*/

make_defs_and_main(k,trees)
int k;
struct production_tree *trees;
{
    char num[5];
    debug("begin make_defs_and_main()",1);

    sprintf(num,"%d",k);
    out_parser("\n# define LL_k      ",0);
    out_parser(num,0);
    out_parser("\n\ncompile()\n",0);
    out_parser("\n",4);
    out_parser(trees->nterm_name,4);
    out_parser("\n",4);
    out_parser("\n",4);
    out_parser("if(LL_lookahead(1)!= END_OF_FILE)\n",4);
    out_parser("LL_syntax_error(LL_text);\n",4);
    out_parser("\n",4);

    debug("end make_defs_and_main()",1);
}

```

```

# include "struct.h"

/*
NAME : make_empty
PURPOSE : to output the code for an action if one exists, else do nothing
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : out_parser
ARGUMENTS PASSED IN :
    pointer to subtree
    current value of the code indentation
VALUE RETURNED : none
LOCAL VARIABLES :
    num - temporary place to hold the string value of a number before
          output
*/

make_empty(tree, indent)
struct tree_node *tree;
int indent;
{
    char num[10];
    if (tree->action_id != 0)
    {
        out_parser("LL", indent);
        sprintf(num, "%d", tree->action_id);
        out_parser(num, indent);
        out_parser("(set_begin);\n", indent);
    }
}

```

```

# include "struct.h"

/*
NAME : make_lexical_analyzer
PURPOSE : to control the creation and output of the lexical analyzer
CALLED BY : main
CALLS : convert_NDFSA_DFSA, make_reg_expr_code
ARGUMENTS PASSED IN : pointer to NDFSA
VALUE RETURNED : boolean, was the lexical analyzer output or not
LOCAL VARIABLES : none
*/

int
make_lexical_analyzer(ndfsa)
struct FSA *ndfsa;
{
    struct FSA *convert_NDFSA_DFSA();
    debug("begin make_lexical_analyzer()",1);
    if (ndfsa != NULL)
    {
        convert_NDFSA_DFSA(ndfsa);
        make_reg_expr_code(ndfsa);
        debug("end make_lexical_analyzer()",1);
        return(TRUE);
    }
    else
    {
        debug("end make_lexical_analyzer()",1);
        return(FALSE);
    }
}

```



```

# include "struct.h"

/*
NAME : make_non_terminal
PURPOSE : to output the code for a non-terminal node in a tree
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : out_parser
ARGUMENTS PASSED IN :
    pointer to subtree
    current indentation of parser code
VALUE RETURNED : none
LOCAL VARIABLES :
    k - character string to hold the value of a number before output
*/

make_non_terminal(tree, indent)
struct tree_node *tree;
int indent;
{
    char k[100];
    debug("begin make_non_terminal()", 1);

    out_parser(tree->left.non_term->nterm_name, indent);
    out_parser("{", indent);
    sprintf(k, "%d", tree->right.ref_num);
    out_parser(k);
    out_parser(");\n", indent);
    debug("end make_non_terminal()", 1);
}

```

```

# include "struct.h"

/*
NAME : make_proc_nonterm
PURPOSE : to output the code for the beginning and end of the subroutine
          for a non_terminal
CALLED BY : make_parser
CALLS : out_parser, make_alternation, make_closure_plus, make_concat,
        make_option, make_non_terminal, make_terminal, make_empty
ARGUMENTS PASSED IN : pointer to a production tree
VALUE RETURNED : none
LOCAL VARIABLES :
    print_end_set - remembers whether to print the end set code
                    at this point based on whether an alternation was seen or not
*/

make_proc_nonterm (prod)
struct production_tree *prod;
{
    int print_end_set;
    print_end_set = TRUE;
    debug("begin make_proc_nonterm ()",1);

    out_parser(prod->nterm_name,0);
    out_parser("(LL_reference)\n",0);
    out_parser("int LL_reference;\n",0);
    out_parser("\n",4);
    out_parser("int *set_begin,*first_closure;\n",4);
    out_parser("int *LL_begin_set(),*LL_begin_closure();\n",4);
    out_parser("first_closure = NULL;\n",4);
    out_parser("set_begin = LL_begin_set();\n",4);
    switch(prod->tree_head->node_type)
    {
        case alternation:
            make_alternation (prod->tree_head,4,FALSE);
            /* end-set printed under each alternative node */
            print_end_set = FALSE;
            break;
        case closure :
            case plus :
                make_closure_plus(prod->tree_head,4);
                break;
        case concat :
            make_concat (prod->tree_head,4);
            break;
        case option :
            make_option (prod->tree_head,4);
            break;
        case non_terminal:
            make_non_terminal(prod->tree_head,4);
            break;
        case terminal :
            make_terminal (prod->tree_head,4);
            break;
        case empty :
            make_empty (prod->tree_head,4);
            break;
    }
    if (print_end_set)

```

```

(
  out_parser("LL_end_set(set_begin,FALSE,first_closure,".4);
  out_parser(last_was_closure(prod->tree_head),4);
  out_parser(");\n",4);
);
  out_parser(")\n",4);
  out_parser("\n",0);
  debug("end  make_proc_nonterm ()",1);
)

```

```

# include "struct.h"
/*****
*/
NAME : make_option
PURPOSE : to output parser code for the option node
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : out_parser, pr_lookahead_if, make_alternation, make_closure_plus,
            make_concat, make_option, make_non_terminal, make_terminal,
            make_empty, last_was_closure
ARGUMENTS PASSED IN :
            pointer to subtree
            current indentation level
VALUE RETURNED : none
LOCAL VARIABLES :
            print_end_set - remembers whether to print the end set based on whether
                           the child was alternation
*/

make_option(tree, indent)
struct tree_node *tree;
int indent;
{
    int print_end_set;
    char *last_was_closure();
    debug("begin make_option()", 1);

    print_end_set = TRUE;
    out_parser("if ( first_closure == NULL )\n", indent);
    out_parser("first_closure = LL_begin_closure();\n", indent+5);
    out_parser("else LL_begin_closure();\n", indent);
    pr_lookahead_if(tree->left.child->first_set, tree->K, indent);
    out_parser("{\n", indent+4);
    out_parser("int *set_begin, *first_closure;\n", indent+4);
    out_parser("first_closure = NULL;\n", indent+4);
    out_parser("set_begin = LL_begin_set();\n", indent+4);
    switch(tree->left.child->node_type)
    {
        case alternation:
            make_alternation (tree->left.child, indent+4, TRUE);
            /* an end-set is printed under each alternative node */
            print_end_set = FALSE;
            break;
        case closure :
            break;
        case plus :
            make_closure_plus(tree->left.child, indent+4);
            break;
        case concat :
            make_concat (tree->left.child, indent+4);
            break;
        case option :
            make_option (tree->left.child, indent+4);
            break;
        case non_terminal:
            make_non_terminal(tree->left.child, indent+4);
            break;
        case terminal :
            break;
    }
}

```

```

        make_terminal (tree->left.child, indent+4);
        break;
    case empty :
        make_empty (tree->left.child, indent+4);
        break;
    };
    if (print_end_set)
    {
        out_parser("LL_end_set(set_begin, TRUE, first_closure, ",
            indent+4);
        out_parser(last_was_closure(tree->left.child), indent+4);
        out_parser(");\n", indent+4);
        out_parser(");\n", indent+4);
    };
    debug("end make_option()", 1);
}

/*****
*/
NAME : pr_lookahead_if
PURPOSE : prints the if statement on those nodes that require lookahead
CALLED BY : make_alternation, make_closure_plus, make_option
CALLS : out_parser
ARGUMENTS PASSED IN :
    pointer to a first set
    value of k
    current indentation level
VALUE RETURNED : none
LOCAL VARIABLES :
    i - counts the tokens to insure lookahead is only at k
    t - goes through the first items in a first set
    s - goes through the tokens in a first item
    inum - temporary storage for a number before output
*/

pr_lookahead_if(first, k, indent)
struct first_follow *first;
int k, indent;
{
    int i;
    struct first_follow *t;
    struct token *s;
    char inum[5];

    /* for each first item in a first set */
    for (t=first; t!=NULL; t=t->next_set)
    {
        if (t==first)
            out_parser("if ( ", indent);
        else
            out_parser("|| ", indent+4);
        /* for each token in a first item */
        for (i=1, s=t->first_token; i<=k && s!=NULL; i++, s=s->next_token)
        {
            if (s==t->first_token)
                out_parser("{ ", indent);
            else
                out_parser(" &&\n", indent);

```

```

    if (s->ref_number != 0)
    {
        out_parser("LL_reference == ", indent+9);
        sprintf(inum, "%d", s->ref_number);
        out_parser(inum);
        out_parser(" &&\n", indent);
    };

    out_parser("LL_lookahead(", indent+9);
    sprintf(inum, "%d", i);
    out_parser(inum, indent);
    out_parser(") == ", indent);
    out_parser(s->term_node->term_name, indent);
};

if (t->next_set != NULL)
    out_parser("(")\n", indent);
else
    out_parser(")\n", indent);
};

if (first == NULL)
    out_parser("if (1)\n", indent);
}

/*****
*/
NAME : last_was_closure
PURPOSE : to determine if the last right_hand node in a set is a closure
          (closure, closure_plus, option)
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : none
ARGUMENTS PASSED IN : pointer to a subtree
VALUE RETURNED : character string, either "TRUE" or "FALSE"
LOCAL VARIABLES :
    true - string constant for "TRUE"
    false - string constant for "FALSE"
*/

char *
last_was_closure(tree)
struct tree_node *tree;
{
    static char *true = "TRUE";
    static char *false = "FALSE";

    for (; tree->node_type == concat; tree = tree->right.right_child);
    if (tree->node_type == closure
        || tree->node_type == option
        || tree->node_type == plus)
        return(true);
    else
        return(false);
}

```

```

#include "struct.h"
/*****
/*
NAME : make_parser
PURPOSE : to control the entire output of parser code
CALLED BY : main
CALLS : find_K, make_defs_and_main, make_proc_nonterm
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : integer value of biggest k
LOCAL VARIABLES :
    k - temporary storage for k
    i - pointer to production tree being worked on
*/

int
make_parser(trees)
struct production_tree *trees;
{
    int k;
    struct production_tree *i;
    debug("begin make_parser()",1);

    k = find_K(trees);
    make_defs_and_main(k,trees);
    /* for each production */
    for (i=trees; i!=NULL; i=i->next_tree)
    {
        if (trace)
            printf("\txs\n", i->nterm_name), fflush(stdout);
        make_proc_nonterm(i);
    };
    debug("end
    return(k);
}

/*****
/*
NAME : find_K
PURPOSE : calculates largest k for the entire grammar
CALLED BY : make_parser
CALLS : find_sub_K
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : integer
LOCAL VARIABLES :
    i - temporary integer
    k - holds largest K found until the value is returned
*/

static int
find_K(trees)
struct production_tree *trees;
{
    int i,k;

    k = 0;
    for (i=trees; i!= NULL; i=trees->next_tree)

```

```

    {
        if (K < (l = find_sub_k(trees->tree_head)))
            K = l;
    }
    return(K);
}

/*****
/*
NAME : find_sub_k
PURPOSE : to recursively go through a tree and find largest K
CALLED BY : find_K
CALLS : find_sub_k
ARGUMENTS PASSED IN : pointer to a subtree
VALUE RETURNED : integer value of largest k found
LOCAL VARIABLES :
    i - holds value of left child
    j - holds value of right child
*/

static int
find_sub_k(node)
struct tree_node *node;
{
    int i,j;

    i = j = 0;
    switch (node->node_type)
    {
        case closure:
        case plus:
        case option:
            i = find_sub_k(node->left_child);
            break;
        case alternation:
        case concat:
            i = find_sub_k(node->left_child);
            j = find_sub_k(node->right_child);
            break;
        case non_terminal:
        case terminal:
            break;
    }
    if (i < j)
        i = j;
    if (i < node->K)
        return(node->K);
    else
        return(i);
}
*****/

```



```

# include "struct.h"

/*
NAME : make_terminal
PURPOSE : to produce code for a terminal node
CALLED BY : make_alternation, make_closure_plus, make_concat,
            make_proc_nonterm, make_option
CALLS : out_parser
ARGUMENTS PASSED IN :
            pointer to subtree
            current indentation value for output code
VALUE RETURNED : none
LOCAL VARIABLES :
            num - temporary storage location for character value of a number
*/

make_terminal(tree, indent)
struct tree_node *tree;
int indent;
{
    char num[10];
    debug("begin make_terminal()", 1);

    out_parser("if (", indent);
    out_parser(tree->left.term_ptr->term_name, indent);
    out_parser(" != LL_get_token())\n", indent);
    out_parser("LL_syntax_error(LL_text);\n", indent+4);
    out_parser("LL_new_entry(", indent);
    out_parser(tree->left.term_ptr->term_name, indent);
    out_parser(");\n", indent);
    debug("end make_terminal()", 1);
}

```

```

/* NAME : (Memory manager)
PURPOSE : to provide two entry points.
           malloc - returns a pointer to area of size 'nbytes'
           free - puts an area back onto the free list
These routines were created to increase the size of the blocks
retrieved in 'morecore', to decrease the fragmentation of the
program's very small data areas because the area grabbed by
'morecore' is relatively small. It now grabs 10K chunks.

These routines were taken directly from The_C_Reference_Manual
by Kernighan and Plauer.
*/

#include <stdio.h>
typedef int ALIGN;
union header {
    struct {
        union header *ptr;
        unsigned size;
    }
    s;
    ALIGN x;
} ;

typedef union header HEADER;

static HEADER base;
static HEADER *allocp = NULL;

char *
malloc(nbytes)
    unsigned nbytes;
{
    HEADER *morecore();
    HEADER *p,*q;
    int nunits;
    nunits = 1 + (nbytes+sizeof(HEADER)-1)/sizeof(HEADER);
    if ((q = allocp) == NULL) {
        base.s.ptr = allocp = q = &base;
        base.s.size = 0;
    }
    for (p=q->s.ptr; !q=p,p=p->s.ptr) {
        if (p->s.size >= nunits) {
            if (p->s.size == nunits)
                q->s.ptr = p->s.ptr;
            else {
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            allocp = q;
            return((char *) (p+1));
        }
        if (p == allocp)
            if ((p=morecore(nunits)) == NULL)
                return(NULL);
    }
}

```

```

)
# define NALLOC 10240

static HEADER *morecore(nu)
unsigned nu;
{
    char *sbrk();
    char *cp;
    HEADER *up;
    int rnu;

    rnu = NALLOC * ((nu+NALLOC-1)/NALLOC);
    cp = sbrk(rnu*sizeof(HEADER));
    if ((int)cp == -1) return(NULL);
    up = (HEADER *)cp;
    up->s.size = rnu;
    free((char *) (up+1));
    return(allocp);
}

free(ap)
char *ap;
{
    HEADER *p,*q;
    p = (HEADER *)ap-1;
    for (q=allocp;!(p>q&p(q->s.ptr):q=q->s.ptr) {
        if (q)=q->s.ptr && (p>q || p(q->s.ptr)) break;
    }
    if (p->s.size == q->s.ptr){
        p->s.size += q->s.ptr->s.size;
        p->s.ptr = q->s.ptr->s.ptr;
    }
    else {
        p->s.ptr = q->s.ptr;
    }
    if (q->s.size == p) {
        q->s.size += p->s.size;
        q->s.ptr = p->s.ptr;
    }
    else {
        q->s.ptr = p;
    }
    allocp = q;
}

```

```

# include "struct.h"

static int newline = TRUE;

/*
NAME : out_parser
PURPOSE : to output to the parser the character string input, with the
appropriate indentation
CALLED BY : make_alternation, make_closure_plus, make_concat,
make_defs_and_main, make_empty, make_non_terminal,
make_proc_nonterm, make_option, pr_lookahead_if,
make_terminal
CALLS : none
ARGUMENTS PASSED IN :
character string
indentation
VALUE RETURNED : none
LOCAL VARIABLES :
newline - global to this routine, remembers whether the last
character on the previous call was a newline
*/
i - count through the indent in a loop

out_parser(line, indent)
char *line;
int indent;
{
    int i;

    if (newline)
        for (i=0; i<indent; i++)
            fprintf(tmp_parser, " ");
    fprintf(tmp_parser, "%s", line);
    if ('\n' == line[strlen(line)-1])
        newline = TRUE;
    else
        newline = FALSE;
}

```

```

# include "struct.h"
/*****
/*
NAME : output_token_list
PURPOSE : print the token list on the "output" file
CALLED BY : YACC file
CALLS : none
ARGUMENTS PASSED IN : pointer to head of token list
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
output_token_list(head)
struct term_list *head;
{
    debug("begin output_token_list()",1);
    fprintf(output,"\nTOKEN LIST\n\n");
    for (;head!=NULL;head=head->next_term)
    {
        fprintf(output,"%s\t%d\n",head->term_name,head->term_num);
    };
    fprintf(output,"\nEND TOKEN LIST\n\n");
    debug("end output_token_list()",1);
}
/*****
/*
NAME : output_fsa
PURPOSE : to print the FSA on the "output" file
CALLED BY : main
CALLS : pchar
ARGUMENTS PASSED IN : pointer to FSA
VALUE RETURNED : none
LOCAL VARIABLES :
    t - pointer to a transition
    INSIDE - remembers whether a set of transitions all have the same
             to-state
    i - integer to loop through the states
*/
output_fsa(fsa)
struct FSA *fsa;
{
    struct trans_type *t;
    int i,INSIDE;
    debug("begin output_fsa()",1);
    fprintf(output,"\nSTATE LIST\n\n");
    for (;fsa!=NULL;fsa=fsa->next_state)
    {
        fprintf(output,"%4d : %3d\n",fsa->state_num,fsa->action_id);
        if (fsa->EMPTY_transition != NULL)
        {
            fprintf(output,"    EMPTY =");
            for (t=fsa->EMPTY_transition;
                t != NULL;

```

```

t = t->next_trans)
{
    fprintf(output, " %d", t->to_state->state_num);
};
fprintf(output, "\n");
};

for (i=0; i<MAX_CHAR_SET; i++)
{
    if (fsa->transition[i] != NULL)
    {
        fprintf(output, " %s", pchar(i));
        for (t=fsa->transition[i]; t != NULL; t=t->next_trans)
        {
            fprintf(output, " %d", t->to_state->state_num);
        };
        fprintf(output, "\n");
    };
};

fprintf(output, "\nEND STATE LIST\n");
debug("end output_fsa()", 1);
}

/*****
*/
NAME : output_prods
PURPOSE : to output the production trees on the "output" file
CALLED BY : main
CALLS : output_node
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

output_prods(tree)
struct production_tree *tree;
{
    debug("begin output_prod()", 1);
    fprintf(output, "\nPRODUCTION TREES\n\n");
    for (; tree != NULL; tree = tree->next_tree)
    {
        fprintf(output, "*****\n\n%s\n", tree->nterm_name);
        output_node(tree->tree_head, 1);
        fprintf(output, "\n");
    };
    fprintf(output, "\nEND PRODUCTION TREES\n\n");
    debug("end output_prod()", 1);
}

/*****
*/
NAME : output_node
PURPOSE : to output the value of a particular tree node on the "output" file
CALLED BY : print_prod
CALLS : output_node, output_f_set
ARGUMENTS PASSED IN :

```

```

        pointer to subtree
        current indentation of the print
    VALUE RETURNED : none
    LOCAL VARIABLES :
        c - character string to hold the representation of which type of node
            this node is
        i - integer for counting indents in loops
*/

static
output_node(tree,indent)
struct tree_node *tree;
int indent;
{
    char *c;
    int i;
    if (tree==NULL)return;
    switch(tree->node_type)
    {
        case empty      : c="empty"
                           : break;
        case plus       : c="+"
                           : break;
        case closure    : c="*"
                           : break;
        case option     : c="?"
                           : break;
        case concat     : c="o"
                           : break;
        case alternation : c="|"
                           : break;
        case non_terminal : c=tree->left.non_term->nterm_name; break;
        case terminal   : c=tree->left.term_ptr->term_name; break;
    };

    for (i=0; i<indent; fprintf(output, " "), i++);
    fprintf(output, "%s", k = %d", c, tree->K);
    if (tree->node_type == non_terminal)
        fprintf(output, " ref = %d\n", tree->right.ref_num);
    else
        fprintf(output, "\n");
    for (i=0; i<indent; fprintf(output, " "), i++);
    fprintf(output, " First Set\n");
    output_f_set(tree->first_set, indent);
    for (i=0; i<indent; fprintf(output, " "), i++);
    fprintf(output, " Follow Set\n");
    output_f_set(tree->follow_set, indent);

    switch(tree->node_type)
    {
        case closure :
        case plus :
        case option :
            output_node(tree->left.child, indent+1); break;
        case alternation :
        case concat :
            output_node(tree->left.left_child, indent+1);
            output_node(tree->right.right_child, indent+1); break;
        default :
            break;
    }
}

/*****

```

```

/*
NAME : output_f_set
PURPOSE : to output a first/follow set on the "output" file
CALLED BY : output_node
CALLS : strlen
ARGUMENTS PASSED IN :
    pointer to first/follow set
    current indentation of output
VALUE RETURNED : none
LOCAL VARIABLES :
    i - Integer to use in loops
    space - remembers how far over the print is for subsequent lines
    t - used to go through tokens in loops
    num - temporary storage for character representation of a number
*/

static
output_f_set(f,indent)
struct first_follow *f;
int indent;
{
    int i,space;
    struct token *t;
    char num[10];

    for (;f!= NULL;f = f->next_set)
    {
        for (i=0;i<indent;fprintf(output,"%i ",i++);
            fprintf(output,"- ");
            space = indent*2+6;
            for (t=f->first_token;t!=NULL;t=t->next_token)
            {
                sprintf(num,"%d",t->ref_number);
                if (space + strlen(t->term_node->term_name)
                    + strlen(num) + 3 > 79)
                {
                    fprintf(output,"\n");
                    space = indent*2+6;
                    for (i=0;i<space;fprintf(output," "),i++);
                };
                fprintf(output,"%s(%s) ",
                    t->term_node->term_name,num);
                space = space + strlen(t->term_node->term_name)
                    + strlen(num) + 3;
            };
            fprintf(output,"\n");
        };
    }
}

```



```

# include "struct.h"
/*****
/*
NAME : remove_ff_sets
PURPOSE : to remove first and follow sets from all production tree nodes
CALLED BY : decorate
CALLS : sub_remove
ARGUMENTS PASSED IN : pointer to production trees
VALUE RETURNED : none
LOCAL VARIABLES :
    p - temporary pointer to a production tree in the loop
*/
remove_ff_sets(prods)
struct production_tree *prods;
{
    struct production_tree *p;
    debug("begin remove_ff_sets()",1);
    for (p=prods; p!= NULL; p=p->next_tree)
    {
        sub_remove(p->tree_head);
    };
    debug("end remove_ff_sets()",1);
}
/*****/

/*****/
/*
NAME : sub_remove
PURPOSE : to recursively go through a tree and remove first and follow sets
CALLED BY : remove_ff_sets
CALLS : release_ff_set, sub_remove
ARGUMENTS PASSED IN : pointer to subtree
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
static
sub_remove(tree)
struct tree_node *tree;
{
    debug("begin sub_remove","",0);
    release_ff_set(tree->first_set);
    release_ff_set(tree->follow_set);
    tree->first_set = tree->follow_set = NULL;
    switch(tree->node_type)
    {
        case empty :
        case terminal :
        case non_terminal :
            debug("empty/terminal/non_terminal",1);
            break;
        case concat :
        case alternation :
            debug("concat/alternation",1);
    }
}

```

```

sub_remove(tree->left.left_child);
sub_remove(tree->right.right_child);
break;
case option :
case plus :
case closure :
    debug("option/closure",1);
    sub_remove(tree->left_child);
    break;
};
debug("end sub_remove()",1);

```

```

# include "struct.h"

static char *space = " ";

/* LEX ACTION PRINT ROUTINES */
/*****

/* NAME : begin_action
PURPOSE : prints code for beginning of a lex action
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : action number
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

begin_action(num)
int num;
{
    fprintf(tmp_lex_action,"%16s %d :\n",space,num);
}

/*****

/* NAME : end_action
PURPOSE : ends a lex action code sequence
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

end_action()
{
    fprintf(tmp_lex_action,"%20sbreak;\n",space);
}

/*****

/* NAME : action_code
PURPOSE : print lex action code
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : character string for code
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

action_code(code)
char *code;
{
    for (;*code == '\n' || *code == '\t' || *code == ' ';code++);

```

```

        fprintf(tmp_lex_action,"%20s%s\n",space,code);
    }

/* PRODUCTION ACTION PRINT ROUTINES */

/*****

/*
NAME : begin_production_action
PURPOSE : output beginning code for a production action
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : action number
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

begin_production_action(num)
int num;
{
    fprintf(tmp_prod_action,"LL_%d(LL_set)\nint *LL_set;\n",num);
    fprintf(tmp_prod_action,"{\n    LL_new_entry(0);\n");
}

/*****

/*
NAME : end_production_action
PURPOSE : outputs ending code for a production action
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

end_production_action()
{
    fprintf(tmp_prod_action,"\n    }\n\n");
}

/*****

/*
NAME : production_action_code
PURPOSE : to output the code for production actions
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : character string for code
VALUE RETURNED : none
LOCAL VARIABLES : none
*/

production_action_code(code)
char *code;
{
    fprintf(tmp_prod_action,"%s",code);
}

```

```

*VARIABLES AND SUBROUTINE OUTPUT CODE*/
/*****
/*
NAME : variable_subroutine_code
PURPOSE : to accept var/subr decl code and output it
CALLED BY : VACC input
CALLS : none
ARGUMENTS PASSED IN : character string code
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
variable_subroutine_code(code)
char *code;
{
    fprintf(var_sub_code, "%s\n", code);
}
/*****
/*
NAME : token_def_out
PURPOSE : to output one define statement for a terminal definition
CALLED BY : VACC input
CALLS : strlen
ARGUMENTS PASSED IN : pointer to a token
VALUE RETURNED : none
LOCAL VARIABLES : none
*/
token_def_out(token)
struct term_list *token;
{
    fprintf(token_def_code, "# define %s\t", token->term_name);
    if (strlen(token->term_name) < 7) fprintf(token_def_code, "\t");
    if (strlen(token->term_name) < 15) fprintf(token_def_code, "\t");
    fprintf(token_def_code, "%d\n", token->term_num);
}

```

```

# include "struct.h"
static struct production_tree *top=NULL, *last=NULL;
/*****
/*
NAME : get_top_prod_list
PURPOSE : since pointers to the top and last production created are
maintained here, the last thing YACC does is get a pointer to
the first production. This routine provides it.
CALLED BY : YACC input
CALLS : none
ARGUMENTS PASSED IN : none
VALUE RETURNED : pointer to top of production tree list
LOCAL VARIABLES : none
*/

struct production_tree *
get_top_prod_list()
{
    return(top);
}
/*****
/*
NAME : new_prod_node
PURPOSE : to create a new production tree node
CALLED BY : YACC input
CALLS : malloc, strcpy, sizeof
ARGUMENTS PASSED IN : character string name of non-terminal
VALUE RETURNED : pointer to production node created
LOCAL VARIABLES :
t - temporary pointer to production node being created
*/

struct production_tree *
new_prod_node(name)
char *name;
{
    struct production_tree *t;
    char *strcpy();
    debug("begin new_prod_name()",1);

    t=(struct production_tree *)malloc(sizeof(struct production_tree));
    t->term_name = strcpy(name);
    t->tree_head = NULL;
    t->next_tree = NULL;
    t->ptree_mark = 0;
    if (last == NULL)
        top = last = t;
    else
        last = last->next_tree = t;
    debug("end new_prod_name()",1);
    return(t);
}
/*****

```

```

/*
NAME : new_tree_node
PURPOSE : to create a new sub_tree node
CALLED BY : YACC input
CALLS : malloc, sizeof, is_ID_terminal, find_prod_node
ARGUMENTS PASSED IN :
    character string defining type of node or the identifier
    for the node, if a terminal or non-terminal
    pointer to left child
    pointer to right child
VALUE RETURNED : pointer to new tree
LOCAL VARIABLES :
    t - temporary pointer to tree node being created
*/

struct tree_node *
new_tree_node (node, left_side, right_side)
char node[];
struct tree_node *left_side, *right_side;
{
    struct tree_node *t;
    struct production_tree *find_prod_node();
    struct term_list *is_ID_terminal();
    debug("begin new_tree_node()", 1);

    t = (struct tree_node *) malloc(sizeof(struct tree_node));
    t->first_set = t->follow_set = NULL;
    t->action_id = 0;
    t->K = -1;
    switch (node[0])
    {
        case 'l':
            t->node_type = alternation;
            t->left_child = left_side;
            t->right_child = right_side;
            break;
        case '*':
            t->node_type = closure;
            t->left_child = left_side;
            break;
        case '+':
            t->node_type = plus;
            t->left_child = left_side;
            break;
        case '.':
            t->node_type = concat;
            t->left_child = left_side;
            t->right_child = right_side;
            break;
        case '?':
            t->node_type = option;
            t->left_child = left_side;
            break;
        case '\0':
            t->node_type = empty;
            break;
        default :
            if (t->left_term_ptr == is_ID_terminal(node))
                t->node_type = terminal;
    }
}

```

```

else
{
    t->node_type = non_terminal;
    t->left.non_term = find_prod_node(node);
    if(t->left.non_term == NULL)
        t->left.non_term = new_prod_node(node);
};

debug("end
return(t);
}

new_tree_node()",l);

debug("end
return(t);
}

/*
NAME : find_prod_node
PURPOSE : returns a pointer to a production tree node for an identifier,
if one exists, else null
CALLED BY : new_tree_node, YACC input
CALLS : strcmp
ARGUMENTS PASSED IN : character string defining non-terminal to be searched for
VALUE RETURNED : pointer to node, or NULL if none
LOCAL VARIABLES :
t - temporary pointer to production node for used in loop to find the
one wanted
*/

struct production_tree *
find_prod_node(node)
char *node;
{
    struct production_tree *t;
    debug("begin find_prod_node()",l);
    for (t=top;t!=NULL;t=t->next_tree)
        if (strcmp(node,t->nterm_name))
            {
                debug("end find_prod_node()",l);
                return(t);
            };
    debug("end find_prod_node()",l);
    return(NULL);
}

```



```

# include "struct.h"

static int state_id = 0;

/*****

/*
NAME : new_state
PURPOSE : to create a new state, and attach it to the last one created
CALLED BY : make_deterministic, YACC input
CALLS : malloc, sizeof
ARGUMENTS PASSED IN : pointer to last state created
VALUE RETURNED : pointer to new state
LOCAL VARIABLES :
    t - pointer to new state being created
    i - integer to loop through transitions to make them NULL
*/

struct FSA *
new_state(last_state)
struct FSA *last_state;
{
    struct FSA *t;
    int i;

    t = (struct FSA *)malloc(sizeof(struct FSA));
    if (last_state != NULL)
        last_state->next_state = t;
    t->state_num = ++state_id;
    t->action_id = 0;
    t->EMPTY_transition = NULL;
    for (i=0; i<MAX_CHAR_SET; i++) t->transition[i] = NULL;
    t->next_state = NULL;
    return(t);
}

/*****

/*
NAME : add_transition
PURPOSE : to add a transition from one state to another
CALLED BY : merge_into, reverse_all_transitions, YACC input
CALLS : delete_transition, malloc, sizeof
ARGUMENTS PASSED IN :
    pointer to state from which transition begins
    pointer to state to which transition goes
    character to transition on
VALUE RETURNED : none
LOCAL VARIABLES :
    t - temporary pointer to the transition being created
*/

add_transition(from,to,on_char)
struct FSA *from,*to;
int on_char;
{
    struct trans_type *t;

    delete_transition(from,to,on_char);

```

```

t=(struct trans_type *)malloc(sizeof(struct trans_type));
t->to_state = to;
switch(on_char)
{
    case EMPTY:
        t->next_trans = from->EMPTY_transition;
        from->EMPTY_transition = t;
        break;
    default:
        t->next_trans = from->transition[on_char];
        from->transition[on_char] = t;
        break;
};
}

/*****
/*
NAME : delete_transition
PURPOSE : delete a transition from on state to another
CALLED BY : remove_empty_transitions, add_transition, reverse_all_transitions
CALLS : free
ARGUMENTS PASSED IN :
    pointer to state from which transition occurs
    pointer to state to which transition goes
    character that is the transition
VALUE RETURNED : none
LOCAL VARIABLES :
    t,s - temporary pointers to transitions
*/

delete_transition(from,to,on_char)
struct FSA *from,*to;
int on_char;
{
    struct trans_type *t,*s;
    /* if the first transition is the one to delete */
    switch(on_char)
    {
        case EMPTY:
            t = from->EMPTY_transition;
            if (t==NULL)break;;
            if (t->to_state == to)
            {
                from->EMPTY_transition = t->next_trans;
                free(t);
                t = NULL;
            };
            break;
        default :
            t = from->transition[on_char];
            if (t==NULL)break;;
            if (t->to_state == to)
            {
                from->transition[on_char] = t->next_trans;
                free(t);
                t = NULL;
            };
    }
}

```

```

        break;
    };
    /* otherwise find the transition in the list */
    if (ti=NULL)
    {
        while (t->next_trans != NULL)
        {
            if (t->next_trans->to_state == to)
            {
                s = t->next_trans;
                t->next_trans = s->next_trans;
                free(s);
            }
            else
                t=t->next_trans;
        }
    };
}

/*****
/*
NAME : reverse_all_transitions
PURPOSE : takes the transitions from one state to another, deletes them,
and inserts transitions on all the other characters
CALLED BY : YACC input
CALLS : delete_transition, add_transition
ARGUMENTS PASSED IN :
    pointer to state from which transitions begin
    pointer to state to which transitions go
VALUE RETURNED : none
LOCAL VARIABLES :
    i - integer to loop through the transitions
    FLAG - remembers whether a transition existed on a particular char
*/

reverse_all_transitions(from,to)
struct FSA *from,*to;
{
    int i;
    int FLAG;
    struct trans_type *fr;

    for (i=EMPTY; i<MAX_CHAR_SET; i++)
    {
        FLAG = FALSE;
        switch(i)
        {
            case EMPTY: fr=from->EMPTY_transition; break;
            default : fr=from->transition[i]; break;
        };
        for (; fr != NULL && !FLAG; fr=fr->next_trans)
        {
            if (fr->to_state == to)
            {
                /* if there is one, delete it */
                FLAG = TRUE;
            }
        }
    }
}

```

```
delete_transition(from,to,i);  
};
```

```
};  
/* If there wasn't one, add one */  
if (!FLAG) add_transition(from,to,i);  
};
```

```
)
```

```

/*
NAME : YACC input (entry point yyparse)
PURPOSE : to accept all input to LL
CALLED BY : yacc
CALLS : output_token_list, get_top_prod_list, malloc, sizeof,
strcpy, token_def_out, new_state, add_transition, begin_action,
end_action, action_code, reverse_all_transitions, yerror,
is_id_terminal, find_prod_node, new_prod_node, new_tree_node,
begin_production_action, end_production_action,
production_action_code, variable_subroutine_code
ARGUMENTS PASSED IN : none
VALUE RETURNED : none
LOCAL VARIABLES :
temp_tree - used as loop variable during perusal of a tree
yytext - character array containing last token returned by lex
line_num - integer variable of the current input line number
(maintained by lex)
begin_fsa - holds pointer to complete FSA when done
last_state - pointer to last state created for giving to new_state
stop_state - the state that is the ending state for a particular
sub_fsa
temp_fsa - temporary pointer to a sub_fsa
begin_tree - pointer to first production tree when all are done
token_list - pointer to head of the token list
last_action_id - holds value of the last action id for lex actions
last_prod_action_id - hold same for production actions
num_concat_items - number of items in a set
concat_array[MAX_ITEMS] - array to hold boolean values as to whether
a particular item in a set is a closure or not
begin_concat_array - beginning of the current set in the concat_array
i - temporary integer
*/

%start
%union
{
    complete_parser_definition
    int intval;
    struct term_list *token_val;
    struct FSA *fsa;
    struct tree_node *subtree;
    struct production_tree *ptree;
}

%token
VAR SUB LINE RBRACE QUOTE LBRACE RETURN ANY_CHAR ESCAPED_QUOTE
PERCENT_COLON SEMICOLON ALTERNATE RPAREN LPAREN LBRACK RBRACK
BEGIN_ACTION IDENTIFIER LEX_ACTION_CODE ESCAPED_RBRACK CARAT
DASH ESCAPE GREATER_I GREATER_O OPTION PERIOD ESCAPED_LBRACK
OCTAL_NUMBER ESCAPED_SPACE DOLLAR_DOLLAR DOLLAR_NUMBER PLUS
<intval>
complete_parser_definition
<token_val>
token_definitions
<intval>
lex_code_lines opt_number_control empty
opt_dashed_character one_bracketed_character opt_carat
escaped_bracketed_character octal_number single_char
escaped_lexical_character quoted_character
<fsa>
opt_regular_expression_set regular_expression_set regular_expression
lexical_expression lex_expr_l lex_element lex_sub_element
bracketed_expression bracketed_characters bracketed_character

```

```

Xtype      quoted_string
<intval>
opt_plus action_item action_code action_code_item
production_action_quote production_action_char
Xtype      <subtree>
Xtype      alternate_list alternate_concat_list concat_item
Xtype      <ptree>
Xtype      production_rules production_rule
Xtype      <intval>
XX         variables_subroutines var_sub_lines
complete_parser_definition :
    if (trace)
        printf("Reading token list.\n"), fflush(stdout);
    token_definitions
    {
        output_token_list(token_list);
        debug("done token_definitions", 1);
    }
PERCENT
    {
        debug("found PERCENT = ", 0);
        debug(yytext, 0);
        debug(":", 1);
        if (trace)
            printf("Reading lexical expressions.\n"), fflush(stdout);
    }
opt_regular_expression_set
    {
        debug("done regular_expression_set", 1);
    }
PERCENT
    {
        debug("found PERCENT = ", 0);
        debug(yytext, 0);
        debug(":", 1);
        if (trace)
            printf("Reading production rules.\n"), fflush(stdout);
    }
production_rules
    {
        debug("done production_rules", 1);
        begin_tree = get_top_prod_list();
        if (trace)
            printf("Reading variables and subroutines.\n"), fflush(stdout);
    }
variables_subroutines
    {
        debug("done variables_subroutines", 1);
    }
;

/*BEGIN TOKEN DEFINITIONS*/
token_definitions :
IDENTIFIER
(

```

```

token_list = (struct term_list *)
    malloc(sizeof(struct term_list));
$$ = token_list;
$$->term_name = strcpy(yytext);
$$->term_num = 1;
$$->next_term = NULL;
token_def_out($$);
}

token_definitions
IDENTIFIER
{
    $$=(struct term_list *)malloc(sizeof(struct term_list));
    $$->term_name = strcpy(yytext);
    $$->term_num = $1->term_num+1;
    $$->next_term = NULL;
    $1->next_term = $$;
    token_def_out($$);
}

;

/*BEGIN REGULAR EXPRESSION*/
empty :
{
    $$ = 0;
}

;

opt_regular_expression_set :
empty
{
    begin_fsa = NULL;
};

regular_expression_set
;

regular_expression_set :
{
    begin_fsa = last_state = new_state(NULL);
}
regular_expression
add_transition(begin_fsa,$2,EMPTY);

;

regular_expression_set
regular_expression
{
    add_transition(begin_fsa,$2,EMPTY);
}

;

regular_expression :
lexical_expression
{
    stop_state->action_id = ++last_action_id;
    begin_action(stop_state->action_id);
}

```

```

    }
    lex_code_lines
    {
        end_action();
        $$ = $1;
    }
;

lex_code_lines :
    lex_code_lines
    LEX_ACTION_CODE
    {
        action_code(yytext);
    }
    |
    LEX_ACTION_CODE
    {
        action_code(yytext);
    }
;

lexical_expression :
    lex_expr_1
    |
    lexical_expression
    {
        $$ = stop_state;
    }
    |
    ALTERNATE
    lex_expr_1
    {
        add_transition($1,$4,EMPTY);
        add_transition(stop_state,$<fsa>2,EMPTY);
        $$ = $1;
        stop_state = $<fsa>2;
    }
;

lex_expr_1 :
    lex_element
    |
    lex_expr_1
    {
        $$ = stop_state;
    }
    |
    lex_element
    {
        $$ = $1;
        add_transition($<fsa>2,$3,EMPTY);
    }
;

lex_element :
    lex_sub_element
    {
        temp_fsa = $1;
    }
    |
    opt_number_control
    {

```



```

    $$ = $1;
    }
;
opt_number_control :
    GREATER_0
    {
        add_transition(stop_state,temp_FSA,EMPTY);
        stop_state = temp_FSA;
    }
    |
    GREATER_1
    {
        add_transition (stop_state,temp_FSA,EMPTY);
    }
    |
    OPTION
    {
        add_transition(temp_FSA,stop_state,EMPTY);
    }
    |
    empty
    ;

lex_sub_element :
    LPAREN
    lexical_expression
    RPAREN
    {
        $$ = $2;
    }
    |
    LBRACK
    bracketed_expression
    RBRACK
    {
        $$ = $2;
    }
    |
    PERIOD
    {
        $$=last_state = new_state(last_state);
        stop_state=last_state = new_state(last_state);
        for (i=1;i<MAX_CHAR_SET;i++)
            if (!='\\n') add_transition($$,stop_state,i);
    }
    |
    single_char
    {
        $$ = last_state = new_state(last_state);
        stop_state=last_state = new_state(last_state);
        add_transition($$,stop_state,$1);
    }
    |
    QUOTE
    quoted_string
    QUOTE
    {
        $$ = $2;
    }

```

```

    }
    ;

bracketed_expression :
(
    temp_FSA=last_state = new_state(last_state);
    stop_state=last_state = new_state(last_state);
)
;

opt_carat
bracketed_characters
(
    if ($2)
    (
        reverse_all_transitions(temp_FSA,stop_state);
    );
    $$ = temp_FSA;
)
;

opt_carat :
empty
(
    $$ = FALSE;
)
;

CARAT
(
    $$ = TRUE;
)
;

bracketed_characters :
bracketed_character
)
bracketed_characters
bracketed_character
;

bracketed_character :
one_bracketed_character
opt_dashed_character
(
    if($2 != -1)
    (
        if ($1 <= $2)
        {
            for (i=$1;i<=$2;i++)
                add_transition(temp_FSA,stop_state,i);
        }
        else
        {
            yyerror("left char > right char");
        }
    )
    else
    {
        add_transition(temp_FSA,stop_state,$1);
    }
)
;

```

HD-A138 061

A GENERATOR OF RECURSIVE DESCENT PARSERS FOR LL(K)  
LANGUAGES(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON  
AFB OH SCHOOL OF ENGINEERING G B PAPROTNY DEC 83  
AFIT/GCS/MA/83D-6

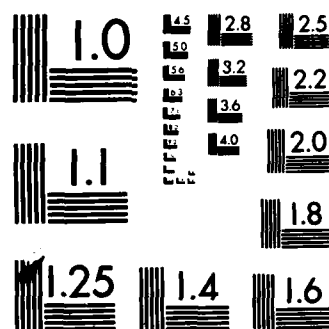
4/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

```

;
opt_dashed_character :
DASH
one_bracketed_character
{
    $$ = $2;
}
|
empty
{
    $$ = -1;
}
;

one_bracketed_character :
octal_number
|
ESCAPED_RBRACK
{
    $$ = ']' ;
}
|
escaped_bracketed_character
|
ANY_CHAR
{
    $$ = yytext[0];
}
;

escaped_bracketed_character :
ESCAPE
ANY_CHAR
{
    switch(yytext[0])
    {
        case 'n' : $$ = '\n'; break;
        case 't' : $$ = '\t'; break;
        case 'r' : $$ = '\r'; break;
        case 'b' : $$ = '\b'; break;
        case 'f' : $$ = '\f'; break;
        default : $$ = yytext[0]; break;
    }
}
|
ESCAPE
ESCAPE
{
    $$ = '\\';
}
|
ESCAPE
DASH
{
    $$ = '-';
}
|
ESCAPE

```

```

CARAT
(
    $$ = '^';
)
;

octal_number :
OCTAL_NUMBER
(
    $$ = 0;
    for (i=2; i<5; i++)
        if (yytext[i]>'7')
            yerror("illegal octal character");
    else
        $$ = $$*8+yytext[i];
)
;

single_char :
octal_number
|
escaped_lexical_character
|
ESCAPED_QUOTE
(
    $$ = "'";
)
|
ESCAPED_LBRACK
(
    $$ = '[';
)
|
ANY_CHAR
(
    $$ = yytext[0];
)
|
ESCAPED_SPACE
(
    $$ = ' ';
)
;

escaped_lexical_character :
ESCAPE
ANY_CHAR
(
    switch(yytext[0])
    {
        case 'n' : $$ = '\n'; break;
        case 't' : $$ = '\t'; break;
        case 'r' : $$ = '\r'; break;
        case 'b' : $$ = '\b'; break;
        case 'f' : $$ = '\f'; break;
    }
)
;

```

```

        default : $$ = yytext[0]; break;
    };

```

```

    |
    | ESCAPE
    | LPAREN
    | (
    | $$ = '(';
    | )
    |
    | ESCAPE
    | RPAREN
    | (
    | $$ = ')';
    | )
    |
    | ESCAPE
    | ALTERNATE
    | (
    | $$ = '|';
    | )
    |
    | ESCAPE
    | GREATER_8
    | (
    | $$ = '>';
    | )
    |
    | ESCAPE
    | GREATER_1
    | (
    | $$ = '>';
    | )
    |
    | ESCAPE
    | OPTION
    | (
    | $$ = '?';
    | )
    |
    | ESCAPE
    | PERIOD
    | (
    | $$ = '.';
    | )
    |
    | ESCAPE
    | ESCAPE
    | (
    | $$ = '\\';
    | )
    |

```

```

quoted_string :
    quoted_character
    (
        $$=last_state = new_state(last_state);
        stop_state=last_state = new_state(last_state);
        add_transition($$,stop_state,$1);
    )
;

```

```

    )
    quoted_string
    quoted_character
    (
        $$ = last_state = new_state(last_state);
        add_transition(stop_state,$$,S2);
        stop_state=$$;
        $$=S1;
    )
;

quoted_character :
    octal_number
    | ESCAPED_QUOTE
    (
        $$ = '...';
    )
    | ANY_CHAR
    (
        $$ = yytext[0];
    )
    | ESCAPE
    ESCAPE
    (
        $$ = '\\';
    )
    | ESCAPE
    ANY_CHAR
    (
        switch(yytext[0])
        {
            case 'n' : $$ = '\n'; break;
            case 't' : $$ = '\t'; break;
            case 'r' : $$ = '\r'; break;
            case 'b' : $$ = '\b'; break;
            case 'f' : $$ = '\f'; break;
            default : $$ = yytext[0]; break;
        }
    )
;

/*BEGIN PRODUCTION RULES*/
production_rules :
    production_rule
    (
        debug("done production_rule",1);
        print_prodiget_top_prod_list();
    )
    | production_rules
    (
        debug("done production_rules",1);
    )
;

```



```

production_rule
{
    debug("done production_rule",1);
    print_prod(get_top_prod_list());
}

;

production_rule :
IDENTIFIER
{
    debug("found IDENTIFIER = '", $);
    debug(yytext, $);
    debug(":", 1);
    if (is_ID_terminal(yytext))
    {
        yerror("'ID is a terminal");
    };
    if (trace)
        printf("\tbegin %s\n", yytext), fflush(stdout);
    $$ = find_prod_node(yytext);
    if ($$ == NULL)
        $$ = new_prod_node(yytext);
}

COLON
{
    debug("found COLON = '", $);
    debug(yytext, $);
    debug(":", 1);
    begin_concat_array = $;
}

alternate_list
{
    debug("done alternate_list", 1);
    print_node($$, $);
}

SEMICOLON
{
    debug("found SEMICOLON", 1);
    if ($<ptree>2->tree_head == NULL)
    {
        $<ptree>2->tree_head = $5;
    }
    else
    {
        $<ptree>2->tree_head =
            new_tree_node("1", $5, $<ptree>2->tree_head);
    };
    debug("endof production_rule", 1);
    print_prod($<ptree>2);
}

;

alternate_list :
{
    $$ = (struct tree_node *) num_concat_items;
}
alternate
{
    debug("done alternate", 1);
}

```

```

num_concat_items = $(intval>1);
print_node($2,$);
$$ = $2;
}
alternate_list
{
    debug("done alternate_list",1);
    $$ = (struct tree_node *) num_concat_items;
}
ALTERNATE
{
    debug("found ALTERNATE",1);
}
alternate
{
    debug("done alternate",1);
    num_concat_items = $(intval>2);
    if ($1->node_type != alternation)
        $$ = new_tree_node("!", $1,$5);
    else
    {
        for(temp_tree = $1;
            temp_tree->right.right_child->node_type==alternation;
            temp_tree = temp_tree->right.right_child);
        temp_tree->right.right_child =
            new_tree_node("!", temp_tree->right.right_child,$5);
        $$ = $1;
    };
    print_node($$,0);
}
;
alternate :
{
    $$ = (struct tree_node *) begin_concat_array;
}
concat_list
{
    debug("done concat_list",1);
    begin_concat_array = $(intval>1);
    $$ = $2;
}
|
empty
{
    debug("done empty",1);
    $$ = new_tree_node("", NULL, NULL);
}
;
concat_list :
{
    num_concat_items = 0;
}
concat_item
{
    debug("done concat_item",1);
    num_concat_items++;
}

```

```

    print_node($2,$);
    $$ = $2;
}

concat_list
{
    debug("done concat_list",1);
    $$ = (struct tree_node *) num_concat_items;
    print_node($1,$);
}

concat_item
{
    debug("done concat_item",1);
    num_concat_items = $(intval)>2 + 1;
    if ($1->node_type != concat)
        $$ = new_tree_node(".", $1,$3);
    else
    {
        for (temp_tree = $1;
             temp_tree->right_child->node_type==concat;
             temp_tree=temp_tree->right_child);
        temp_tree->right_child = new_tree_node(".",
            temp_tree->right_child,$3);
        $$ = $1;
    }
    print_node($$,0);
}

concat_item :
LPAREN
{
    debug("found LPAREN",1);
    $$ = (struct tree_node *) begin_concat_array;
    begin_concat_array += num_concat_items;
}

alternate_list
{
    debug("done alternate_list",1);
}

RPAREN
{
    debug("found RPAREN",1);
    begin_concat_array = $(intval)>2;
    concat_array[num_concat_items+begin_concat_array] = FALSE;
    $$ = new_tree_node(".",
        new_tree_node(".",
            new_tree_node("", NULL, NULL),
            $3),
        new_tree_node("", NULL, NULL));
}

LBRACK
{
    debug("found LBRACK",1);
    $$ = (struct tree_node *) begin_concat_array;
    begin_concat_array += num_concat_items;
}

alternate_list

```

```

(
    debug("done alternate_list",1);
)
}

RBRACK
(
    debug("found RBRACK",1);
    begin_concat_array = $(intval>2;
    concat_array[num_concat_items+begin_concat_array] = TRUE;
    $$ = new_tree_node("?", $3, NULL);
)
}

LBRACE
(
    debug("found LBRACE",1);
    $$ = (struct tree_node *) begin_concat_array;
    begin_concat_array += num_concat_items;
)
}

alternate_list
(
    debug("done alternate_list",1);
)
}

RBRACE
opt_plus
(
    debug("found RBRACE",1);
    begin_concat_array = $(intval>2;
    concat_array[num_concat_items+begin_concat_array] = TRUE;
    if ($6)
        $$ = new_tree_node("+", $3, NULL);
    else
        $$ = new_tree_node("*", $3, NULL);
)
}

IDENTIFIER
(
    debug("found IDENTIFIER",1);
    concat_array[num_concat_items+begin_concat_array] = FALSE;
    $$ = new_tree_node(yytext, NULL, NULL);
)
}

action_item
(
    debug("done action_item",1);
    $$ = new_tree_node("", NULL, NULL);
    concat_array[num_concat_items+begin_concat_array] = FALSE;
    $$->action_id = $1;
)
}

;

opt_plus :
PLUS
(
    $$ = 1;
)
}

|
empty
(
    $$ = 0;
)
}

```

```

;
action_item :
BEGIN_ACTION
(
    debug("found BEGIN_ACTION",1);
    begin_production_action(++last_prod_action_id);
)
action_code
(
    debug("done action_code",1);
)
RBRACE
(
    debug("found RBRACE",1);
    end_production_action();
    $$ = last_prod_action_id;
)
;

action_code :
empty
|
action_code
action_code_item
;

action_code_item :
LBRACE
(
    production_action_code("(");
)
action_code
RBRACE
(
    production_action_code(")");
)
|
QUOTE
(
    production_action_code("\"");
)
production_action_quote
QUOTE
(
    production_action_code("\"");
)
|
ANY_CHAR
(
    production_action_code(yytext);
)
|
RETURN
(
    production_action_code(yytext);
)
|
DOLLAR_DOLLAR

```

```

(
    char k[15];
    debug("found DOLLAR_DOLLAR",1);
    sprintf(k, "(%LL_set+Xd)", num_concat_items+1);
    production_action_code(k);
    debug("done DOLLAR_DOLLAR",1);
)

DOLLAR_NUMBER
(
    char k[30], j[100]; int i;
    j[0] = '\0';
    debug("found DOLLAR_NUMBER = ",0);
    debug(yytext,1);
    sscanf(yytext, "%d[xs]", &i, j);
    if (i > num_concat_items)
        yyerror("dollar number too big");
    if (strcmp(j, "")
        && concat_array[begin_concat_array+1-1] == FALSE)
        sprintf(k, "(%LL_set+Xd)", i);
    else
        if (strcmp(j, ""))
            && concat_array[begin_concat_array+1-1] == TRUE)
            {
                j[strlen(j)-1] = '\0';
                sprintf(k, "(%((int *)(&LL_set+Xd))+xs)", i, j);
            }
        else
            yyerror("illegal use of $n");
    production_action_code(k);
    debug("done DOLLAR_NUMBER",1);
)

production_action_quote :
empty
|
production_action_quote
production_action_char
;

production_action_char :
ESCAPED_QUOTE
(
    production_action_code("\\\\");
)
|
ANY_CHAR
(
    production_action_code(yytext);
)
;

/*BEGIN VARIABLES AND SUBROUTINES*/
variables_subroutines :
empty
(
)

```

```

    |
    | PERCENT
    | var_sub_lines
    | {
    | }
    | ;
    |
    | var_sub_lines :
    | empty
    | {
    | }
    | ;
    |
    | var_sub_lines
    | VAR_SUB_LINE
    | {
    | variable_subroutine_code(yytext);
    | }
    | ;

%%

# include "struct.h"

struct production_tree *get_top_prod_list(), *find_prod_node(), *new_prod_node();
char *strcpy();
struct FSA *new_state();
struct tree_node *new_tree_node(), *temp_tree;

extern char yytext[];
int yylex();
extern int line_num; /*From lex.l*/
static struct FSA *begin_fsa, *last_state, *stop_state, *temp_FSA;
static struct production_tree *begin_tree;
static struct term_list *token_list;
static int last_action_id = 0, last_prod_action_id = 0;
static int i, num_concat_items, concat_array[MAX_ITEMS], begin_concat_array;

/*****
NAME : yacc
PURPOSE :
CALLED BY :
CALLS :
ARGUMENTS PASSED IN :
VALUE RETURNED :
DATA STRUCTURES USED :
GLOBAL VARIABLES :
LOCAL VARIABLES :
*/

yacc(ndfsa, trees)
struct FSA **ndfsa;
struct production_tree **trees;
{
    debug("begin yacc()", 1);
    begin_fsa = NULL;
    begin_tree = NULL;
    yyparse();
}

```

```

*ndfsa = begin_fsa;
*trees = begin_tree;
debug("end yacc()",1);
}

/*****
/*
NAME : is_ID_terminal
PURPOSE : peruse the terminal list and return TRUE or FALSE
CALLED BY : new_tree_node, YACC input
CALLS : none
ARGUMENTS PASSED IN : character string
VALUE RETURNED : boolean
LOCAL VARIABLES :
t - pointer to each item in the terminal list in loop
*/

struct term_list *
is_ID_terminal(node)
char *node;
{
    struct term_list *t;
    debug("begin is_ID_terminal()",1);
    for (t=token_list;t!= NULL; t=t->next_term)
        if(strcmp(node,t->term_name))
        {
            debug("end is_ID_terminal(yes)",1);
            return(t);
        };
    debug("end is_ID_terminal(no)",1);
    return(NULL);
}

/*****
/*
NAME : yyerror
PURPOSE : to abort processing after recognizable error
CALLED BY : decorate, calc_first_set, sub_find_recursion,
find_undefined_nonterm, find_unused_productions, LEX input,
YACC input
CALLS : close_files, delete_temp_files
ARGUMENTS PASSED IN : character string as output message
VALUE RETURNED : none
LOCAL VARIABLES :
uses : line_num - line number from LEX input
yytext - character string of current token from YACC input
*/

yyerror(s)
char *s;
{
    extern int line_num;
    extern char yytext[];
    debug("begin yyerror()",1);

    printf("%s. Token was = '%s'. Input line number = %d\n",

```



```
s.yytext,line_num).fflush(stdout);
fprintf(output,"%s. Token was = '%s'. Input line number = %d\n",
s.yytext,line_num);
close_files();
delete_temp_files();

debug("end yyerror()",1);
exit(1);
}
```

Standard Code File 1  
-----

```
# define TRUE      1
# define FALSE     0
# ifndef LL_MAXCHAR
# define LL_MAXCHAR 255
# endif
```

```
char LL_text[LL_MAXCHAR+1]; /*contains the current token string*/
```

# Standard Code File 2

```

int LL_length;          /*contains the current token's length*/

/* standard output = listing and error messages
   standard input = data to be read in */

/* LEXICAL ANALYZER */

static char LL_area[LL_MAXCHAR+1];

static int LL_lex()
{
    static struct stack_type
    {
        int action;
        char st_char;
    }
    stack [LL_MAXCHAR];
    /*Each element will contain the action_id for a state,
       and the character that came next. the transition to the
       next state will be made on this character. The first
       element will always have state g's action id*/
    int stackplace;      /*points to the last used stack element*/
    int transition_OK;   /*=TRUE while transitions can be made*/
    int next_state;     /*contains next state # after a transition*/
    int i;
    char c;
    while (TRUE) /*will continue to execute until a return() is executed*/
    {
        stackplace = g;
        next_state = g;
        transition_OK = TRUE;
        for (;transition_OK == TRUE;stackplace++)
        {
            stack[stackplace].st_char = c = LL_getc();
            if (c == EOF) stack[stackplace].st_char = c = END_OF_FILE;
            switch(next_state)
            {

```

Standard Code File 3

```
    );  
    /* the last state was the last good transition */  
    do  
    {  
        stackplace--;  
        LL_putc(stack[stackplace].st_char);  
    }  
    while(stackplace > 0 && stack[stackplace].action == 0);  
    if (stackplace == 0 && stack[stackplace].st_char == END_OF_FILE)  
    {  
        LL_area[0] = '\0';  
        return(END_OF_FILE);  
    };  
    if (stackplace == 0)  
        LL_abort("character cannot be parsed");  
    for (i=0; i<stackplace; i++) LL_area[i] = stack[i].st_char;  
    LL_area[i] = '\0';  
    switch(stack[stackplace].action)  
    {
```

Standard Code File 4

```

    );
}

/* SUBROUTINES FOR LEXICAL ANALYZER */

static char LL_input_line[LL_MAXCHAR*2];
static int LL_input_length = 0;
static int LL_curchar = 1;
static int LL_errorchar;

static char LL_getc()
{
    if (LL_curchar >= LL_input_length)
    {
        getline();
        LL_curchar = 0;
        LL_errorchar = 0;
    };
    LL_errorchar++;
    LL_curchar++;
    return(LL_input_line[LL_curchar-1]);
}

static LL_putc(c)
char c;
{
    int i;
    if (LL_curchar == 0)
    {
        if (LL_input_length == LL_MAXCHAR*2)
            LL_abort("put too many chars back. Compiler abort");
        for (i = LL_input_length; i > 0; i--)
            LL_input_line[i] = LL_input_line[i-1];
        LL_input_length++;
        LL_curchar = 1;
    }
    LL_errorchar--;
    LL_curchar--;
    LL_input_line[LL_curchar] = c;
}

static getline()
{
    for (LL_input_length = 0;
         LL_input_length < LL_MAXCHAR*2 &&
         (LL_input_line[LL_input_length] = getchar()) != EOF;
         LL_input_length++)
    {
        if (LL_input_line[LL_input_length] == '\n')
            break;
    };
    if (LL_input_length == LL_MAXCHAR*2)
    {
        LL_errorchar = -1;
    }
}

```

```

LL_abort("next LL_input_line too long");
LL_input_length++;
LL_input_line[LL_input_length] = '\0';
if (LL_input_line[LL_input_length-1] == EOF)
{
    int i;
    for (i=0; i<LL_input_length-1; i++)
        printf("%c", LL_input_line[i]);
    printf("\n");
}
else
{
    printf(LL_input_line);
}

```

Standard Code File 5

```

/* SUBROUTINES FOR PARSER */

static struct LL_table
/* #th entry not used */
int token_num;
char token_name[LL_MAXCHAR];
LL_lookahead_table[LL_k+1];

static int LL_num_lookahead = #;

static int LL_get_token()
{
    int i, token;
    if (LL_num_lookahead)
    {
        token = LL_lookahead_table[i].token_num;
        LL_strcpy(LL_lookahead_table[i].token_name, LL_text);
        for (i=1; i<LL_num_lookahead; i++)
        {
            LL_lookahead_table[i].token_num =
                LL_lookahead_table[i+1].token_num;
            LL_strcpy(LL_lookahead_table[i+1].token_name,
                LL_lookahead_table[i].token_name);
        };
        LL_num_lookahead--;
    }
    else
    {
        token = LL_lex();
        LL_strcpy(LL_area, LL_text);
    };
    LL_length = strlen(LL_text);
    return(token);
}

static int LL_lookahead(n)
int n;
{
    int token;
    if (n > LL_num_lookahead)
    {
        if (n > 1 && LL_lookahead_table[n-1].token_num == END_OF_FILE)
        {
            LL_lookahead_table[n].token_num = END_OF_FILE;
        }
        else
        {
            LL_lookahead_table[n].token_num = token = LL_lex();
            LL_strcpy(LL_area, LL_lookahead_table[n].token_name);
        };
        LL_num_lookahead++;
    }
    else
    {

```

```

        token = LL_lookahead_table[n].token_num;
    };
    return(token);
}

```

/\* SUBROUTINES FOR VALUE STACK IN PARSER \*/

```

# ifndef LL_VAL_STACK
# define LL_VAL_STACK 10000
# endif

static int LL_val [LL_VAL_STACK];
static int LL_cval[LL_VAL_STACK];
static int *LL_top = &(LL_val [0]);
static int *LL_ctop = &(LL_cval[0]);
static int *LL_bound = &(LL_val [LL_VAL_STACK-1]);
static int *LL_cbound = &(LL_cval[LL_VAL_STACK-1]);

static int *
LL_begin_closure()
{
    *LL_top++ = (int) LL_ctop;
    *LL_ctop++ = 0;
    LL_check_stacks();
    return(LL_ctop-1);
}

```

```

static int *
LL_begin_set()
{
    LL_top++;
    LL_check_stacks();
    return(LL_top-1);
}

```

```

static
LL_new_entry(value)
int value;
{
    *LL_top++ = value;
    LL_check_stacks();
}

```

```

static
LL_end_set(set_begin, in_closure, first_closure, last_item_closure)
int *set_begin;
int in_closure;
int *first_closure;
int last_item_closure;
{
    int set_value;
    if (last_item_closure)
    {
        if (*LL_top-1) == 0
            set_value = 0;
        else
            set_value = *(LL_ctop-1);
    }
    else

```



```

        set_value = *(LL_top-1);
        if (first_closure != NULL)
            LL_ctop = first_closure;
        if (!in_closure)
        {
            *LL_ctop++ = set_value;
            LL_top = set_begin;
            *((int *)*(LL_top-1)) = (((int *)*(LL_top-1))+1);
        }
        else
        {
            LL_top = set_begin;
            *LL_top++ = set_value;
        };
        LL_check_stacks();
    }

static LL_check_stacks()
{
    if (LL_top > LL_bound || LL_ctop > LL_cbound)
        LL_abort("LL error. Value stack overflow.");
}

/* GENERAL SUBROUTINES */

LL_abort(str)
char *str;
{
    int i;
    if (LL_error_char)
        for (i=0; i < LL_errorchar; i++)
            printf("%c", LL_error_char[i]);
    printf("%s\n", str);
    fflush(stdout);
    exit(0);
}

static LL_syntax_error(token)
char *token;
{
    char message[LL_MAXCHAR+31];
    sprintf(message, "syntax error. The token was %s.", token);
    LL_abort(message);
}

static
LL_strcpy(t,s)
char *s,*t;
{
    while (*s++ != '\0');
}

LL_print_stacks(set_begin,first_closure)
int *set_begin,*first_closure;
{
    extern int *LL_top,*LL_ctop,LL_val[],LL_cval[];
    int *i,*j,*k;
    printf("\n");
    for (i= &LL_val[0],j= &LL_cval[0],k=0;

```

```

    i<LL_top || j<LL_ctop; i++, j++, k++)
    {
        if (i<LL_top)
        {
            printf("X3d X5d ", k, "1");
            if (set_begin == 1)
                printf("set_begin");
            else
                printf(" ");
        }
        else
        {
            printf(" ");
            printf("1 ");
            printf("X3d X5d ", k, "j");
            if (first_closure == j)
                printf("first_closure");
            else
                printf(" ");
        }
        else
        {
            printf(" ");
            printf("-\n");
        }
        printf("\n");
        fflush(stdout);
    }
}

```

## VITA

George Bernard Paprotny was born on 25 June 1957 in St. Paul, Minnesota. He graduated from high school in Bowie, Maryland in 1974 and attended the University of Maryland in College Park, Maryland, from which he received the degree of Bachelor of Science in Computer Science in June 1978. Upon graduation he was commissioned in the United States Air Force. In October, 1978, he was assigned as a Computer Systems Analyst to the Directorate of Data Automation, Deputy Chief of Staff/Comptroller, Headquarters Air Training Command, Randolph Air Force Base, Texas. He entered the School of Engineering, Air Force Institute of Technology, in June 1982. He is married to the former Janice Gail Hobbs of Minot, North Dakota.

Permanent Address: 2706 Keyport Lane  
Bowie, Maryland 20715

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AD-A138061

## REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT  Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)  AFIT/GCS/MA/83D-6			7a. NAME OF MONITORING ORGANIZATION		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering Air Force Institute of Tech.		6b. OFFICE SYMBOL (If applicable)  AFIT/EN	7b. ADDRESS (City, State and ZIP Code)		
6c. ADDRESS (City, State and ZIP Code)  Wright-Patterson AFB, Ohio 45433			9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	10. SOURCE OF FUNDING NOS.		
8c. ADDRESS (City, State and ZIP Code)			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
11. TITLE (Include Security Classification)  See Box 19.					WORK UNIT 12.
12. PERSONAL AUTHOR(S)  George B. Paprotny, Captain, USAF					
13a. TYPE OF REPORT  MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day)  December 1983	
15. PAGE COUNT  310					
16. SUPPLEMENTARY NOTATION  Approved for public release: IAW AFR 190-17. <i>Lynn E. Lawlis</i> 7 Feb 84 Lynn E. Lawlis Data for Research and Professional Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.			
09	02		compilers programming languages		
19. ABSTRACT (Continue on reverse if necessary and identify by block number)  Title: LL - A GENERATOR OF RECURSIVE DESCENT PARSERS FOR LL(K) LANGUAGES  Thesis Advisor: Captain Patricia K. Lawlis					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT  UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION  UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL  Patricia K. Lawlis, Instructor in Mathematics		22b. TELEPHONE NUMBER (Include Area Code)  (513) 255-4185		22c. OFFICE SYMBOL  AFIT/ENC	

DD FORM 1473, 83 APR

EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED  
SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

A computer program was designed to generate a recursive-descent compiler for LL(k) languages. A grammar specification language was designed using Extended BNF, along with a lexical specification language using regular expressions.

The program uses a well-tested but not formally proven method for determining the maximum possible k for a grammar. It then uses an iterative first and follow set calculation algorithm to determine the actual k for the grammar.



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

**FILMED**

**4-84**

**DTIC**